November 1988

Report No. STAN-CS-88-1231

Thesis

(4)

THE BETA OPERATION: A PARALLEL PRIMITIVE

by

Evan Reid Cohn

Department of Computer Science

Stanford University

Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp Date Jun 30, 1986

1a REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release: distribution unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S): STAN-CS-88-1231	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Stanford University; 6b OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Stanford, California 94305	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: DARPA; 8b OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-85-C-0731
8c. ADDRESS (City, State, and ZIP Code): Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification): The beta operation: A parallel primitive

12 PERSONAL AUTHOR(S): Evan Reid Cohn

13a TYPE OF REPORT	13b TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
		November 1988	92

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The ever-decreasing cost of computer processors has created a great interest in multiprocessor computers. However, along with the increased power that this parallelism brings, comes increased complexity in programming.

One approach to lessening this complexity is to provide the programmer with general purpose parallel primitives that shield him from the structure of the underlying machine. There are two contending goals that must be satisfied when designing primitives. On one hand we would like to make the primitive as general and abstract as possible. The more general a primitive is, the more easily it can be used as a building block for creating complex algorithmic constructs. On the other hand, we want to avoid making the primitive so general that it becomes inefficient.

In *The Connection Machine*, Hillis suggests the *beta operation* as a parallel primitive for his hypercube-based machine. We shall examine the beta operation, demonstrate its efficiency, generalize it in several directions, and show its suitability for general use.

(KP)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	
22a NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

THE BETA OPERATION:
A PARALLEL PRIMITIVE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Evan Reid Cohn
November 1988

Abstract

The ever-decreasing cost of computer processors has created a great interest in multiprocessor computers. However, along with the increased power that this parallelism brings, comes increased complexity in programming.

One approach to lessening this complexity is to provide the programmer with general purpose parallel primitives that shield him from the structure of the underlying machine. There are two contending goals that must be satisfied when designing primitives. On one hand we would like to make the primitive as general and abstract as possible. The more general a primitive is, the more easily it can be used as a building block for creating complex algorithmic constructs. On the other hand, we want to avoid making the primitive so general that it becomes inefficient.

In *The Connection Machine* [Hil85], Hillis suggests the ***beta operation*** as a parallel primitive for his hypercube-based machine. We shall examine the beta operation, demonstrate its efficiency, generalize it in several directions, and show its suitability for general use.

Acknowledgement

It gives me pleasure to acknowledge the enormous and indispensable contribution of my dissertation advisor, Jeffrey Ullman, who through his knowledge and insight has led me to a deeper understanding of the subject. Not only was he an indefatigable source of ideas, he was always available for discussion. Thanks go also to the other two members of my reading committee, Ernst Mayr and Christos Papadimitriou for their support and many helpful suggestions. I would also like to thank Ramsey Haddad, with whom I did the original work on the Beta Operations and C. Gregory Plaxton, with whom I had a number of stimulating conversations. I am grateful to Joseph Pallas and Oren Patashnik, both of whom provided copious LaTeX support. Penultimately, I would like to thank my friends, with whom I shared the ups and downs of graduate school. Above all, I would like to thank my parents, without whom I would surely not have been here.

Much of this research was supported by an NSF fellowship and by a DARPA contract. I am extremely grateful for the research opportunity afforded me.

Contents

List of Figures

Chapter 1

Introduction

The ever-decreasing cost of computer processors has fueled an increased interest in multiprocessor networks and parallel computation. Much work has been performed on producing efficient parallel versions of sequential algorithms. These parallel algorithms present considerably more complex programming requirements than their associated sequential versions.

A standard method of lessening the complexity of sequential programming is to provide the programmer with abstract data structures and operations that shield him from the structure of the underlying machine. The problem of creating an algorithm for a particular problem then divides into two simpler problems, namely, creating an algorithm in terms of certain data structures and implementing the data structures on a particular machine.

The advantages of this method are twofold. The details of how data are to be moved are hidden in the implementation of the data structures. Also, once an algorithm is expressed in terms of data structures and their manipulation, it can be used on any machine.

The motivation is the same for studying parallel primitives. We want to shield the algorithm designer from details of the underlying network. On the one hand, we want to determine the relationship between various parallel primitives and particular networks. On the other hand, we would like to study how various parallel algorithms can be decomposed into a set of primitive operations. The parallel primitive operations can be used as building blocks for creating complex algorithmic constructs, making it possible to reason about and describe algorithms without explicit reference to the networks. This approach has been explored by a number of researchers ([Hoc85], [RBJ88], [Hua85],[Ble87]).

There are two contending goals that must be satisfied when designing a parallel primitive. It is desirable to make the primitive as general as possible, for the more general it is, the more readily it can be employed to create algorithms. However, increasing generality is often accompanied by a loss of efficiency. At some point the primitive becomes highly general and very inefficient. It is of great importance, therefore, to find an optimal balance.

In *The Connection Machine* [Hil85], Hillis proposes the *beta operation* as a suitable parallel primitive for his hypercube-based machine. In the subsequent chapters, we shall examine the beta

operation, demonstrate its efficiency, generalize it in several directions and show that it is suitable for general use.

1.1 Thesis Organization

In Chapter 2, we present two variant formulations of the beta operation and introduce some notation.

In Chapter 3, we study the problem of implementing the beta operation on several multiprocessor networks, namely, the hypercube, mesh, and mesh-of-trees graphs. The hypercube implementation is presented first and serves as a paradigm for the other implementations. AT^2 lower bounds are presented for circuits implementing the beta operation. These bounds are close to the upper bounds realized in the mesh-of-trees implementation.

In Chapter 4, the beta operation is generalized to cover all possible relationships between the number of processors, the number of input pairs and the number of groups represented by these pairs.

In Chapter 5, we re-examine the hypercube implementation of the beta operation from Chapter 3 and consider the implementation of a new primitive that generalizes the beta operation, the multi-prefix operation. We show that by creating a circuit that stores information about the partial computations that arise during the hypercube implementation of the beta operation, we can implement the multi-prefix operation with no increase in complexity. We can use this implementation as a paradigm for implementing the multi-prefix operation efficiently on other networks.

In Chapter 6, we present some applications of the beta operation. We first demonstrate both the efficiency and the generality of the beta operation by simulating complex beta operations using only simpler beta operations, with only a constant factor increase in the time required. We next show a sample application of one of the results established in Chapter 4.

In Chapter 7, we review the results presented and consider open questions.

Chapter 2

The Beta Operation

2.1 Definition

We begin by presenting the definition of the beta operation. As an instance of the beta operation, we are given an associative, commutative binary function F and N pairs, $(g_0, v_0), \ldots, (g_{N-1}, v_{N-1})$. The g_j's should be thought of as *group numbers* and the v_j's as *data*. Let us denote by B_i, the collection of (g, v)-pairs with $g = i$ (for some v). We shall call the set of processors holding these pairs $proc\left(B_i\right)$. Let $G = \{i \mid |B_i| > 0\}$. We shall assume that every processor has a unique binary address of length $\log P$. The processors of $proc\left(B_i\right)$ are ordered by their binary addresses.

For a two-argument function F and a list of values $C = [c_0, \ldots, c_m]$, let us define the *F-reduction of C* as the natural reduction. That is

- $F/[c_0] = c_0$;
- $F/[c_0, \ldots, c_m] = F(F/[c_0, \ldots, c_{m-1}], c_m)$

If l_i is the list of v-values from B_i, then the beta operation computes the output values $y_i = F/l_i$ for $i \in G$ and produces pairs (i, y_i).

For simplicity, we shall hereafter refer to $|G|$ as S. We shall call a beta operation with N input pairs, P processors, and S output groups, $\beta(N, P, S)$. If the values of N and P are understood, we shall call the operation simply $\beta(S)$. We shall refer to two variant formulations of the beta operation. We shall term them $\beta_1(S)$ and $\beta_2(S)$. If the (i, y_i) pairs end up stored, sorted by g-value, in the S highest-address (lowest-address) processors, we shall call the operation ${\beta_1}^+(S)$ (respectively, ${\beta_1}^-(S)$). We shall call the operation simply $\beta_1(S)$ if the sign of the superscript is not important. The description of the $\beta_2(S)$ operation is identical except that the y_i's are distributed to the processors so that at the completion, all the members of $proc\left(B_i\right)$ hold y_i.

Example 2.1.1 *Let F be '+' and N be 5. Let the (g, v)-pairs in the input be:*

proc 1: (9,6) *proc* 2: (2,4) *proc* 3: (3,4) *proc* 4: (9,5) *proc* 5: (2,5)

Performing $\beta_1{}^+(S)$ *on these pairs produces the output:*

proc 3: (2, 9) *proc* 4: (3, 4) *proc* 5: (9, 11)

Example 2.1.2 *Let* F *be '+' and* N *be 5. Let the* (g, v)*-pairs in the input be:*

proc 1: (9, 6) *proc* 2: (2, 4) *proc* 3: (3, 4) *proc* 4: (9, 5) *proc* 5: (2, 5)

Performing $\beta_2(S)$ *on these pairs produces the output:*

proc 1: (9, 11) *proc* 2: (2, 9) *proc* 3: (3, 4) *proc* 4: (9, 11) *proc* 5: (2, 9)

We shall use the word model of computation in our multiprocessor networks. In this model, every processor has a local memory with words of length $O(\log N)$; the standard set of ALU operations can be performed in constant time on these words. In our model of the hypercube, each processor sends or receives a single message in each time step.

2.2 Notation

For many of the algorithms presented below, we shall need a simple way to refer to sets of processors. For convenience in reference, we shall view processor addresses as being composed of several fields, (e.g., $u_k v_k w_k x_k$). We shall use the convention that the capitalized versions of the field names (e.g., V_k for field v_k) will represent a value for that field, that is, a fixed string of the appropriate length. We shall also use the convention that a value with an underscore beneath it (e.g., $\underline{V_k}$) will represent the set of all possible strings of the appropriate length. Thus, $U_k V_k W_k \underline{X_k}$ represents the $2^{|X_k|}$ processor addresses $U_k V_k W_k X_k$ for $0 \leq X_k < 2^{|X_k|}$, and U_k in field u_k, V_k in field v_k, and W_k in field w_k.

For simplicity, we shall assume that S is a power of 2. It is true that $2^{j-1} < S \leq 2^j$ for some j. If we assume that S is really 2^j, the algorithms of the following chapters will work with the same asymptotic time complexities. We shall call $\log S$, s. We shall call $S-1$, θ, and $S-2$, ϕ. Note that the binary representation of the former is a string of s 1's and the binary representation of the later is a string of $s-1$ 1's followed by a zero. We shall use the convention that a digit, d, with a superscript, k, (e.g., θ^k) will be shorthand for the number whose binary representation is a string of k consecutive d's.

We shall call the operation of sorting N items on a P-node network, SORT(N, P). The identity of the network in question will be understood by context. We shall call the minimum time required to perform SORT(N, P) on this network, $T_{\text{SORT}}(N, P)$.

Chapter 3

Implementing the Beta Operation

3.1 Overview

In this chapter, we shall consider the problem of implementing $\beta_1{}^+(N, N, S)$ on several multiprocessor networks.[1] In Section 3.2, we present an efficient hypercube implementation of $\beta_1{}^+(N, N, S)$. In Section 3.3, we show how $\beta_1{}^+(N, N, S)$ can be implemented on a mesh-of-trees network, using the hypercube implementation as a paradigm. Additionally, the hypercube implementation can be used as a paradigm for implementing $\beta_1{}^+(N, N, S)$ on the butterfly and shuffle-exchange networks.

Both of the implementations that we present assume that the value of S is known. In Section 3.4, we show that the same time bounds can be achieved without knowing S beforehand. Finally, in Section 3.5 we present an AT^2 lower bound that is only a few factors of log below the upper bound presented in Section 3.3.

3.2 Efficient Hypercube Implementation

We start by considering the efficient hypercube implementation of $\beta_1{}^+(N, N, S)$. Let $N = 2^n$ and $S = 2^s$. We shall call n/s, D. The case where s does not divide n evenly can be treated by trivially modifying the algorithm to follow. The processor addresses will be viewed as being composed of D base-S digits. We shall view these base-S addresses as being composed of four fields, $u_k v_k w_k x_k$, where k is the number of the step we are performing. The lengths of fields u_k and w_k are functions of k. The fields are composed as shown in Figure 3.1. The implementation will require time

$$O(\log N + T_{\text{SORT}}(S, S)).$$

3.2.1 The General Step

In each step of this algorithm, we shall conceptually break the hypercube into a number of smaller subcubes. In each step of Phases 1 and 2, we shall apply the three *stages* shown below to all of the

[1] Much of the material of this chapter was first presented in [CH86].

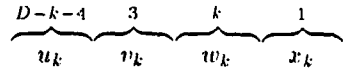

Figure 3.1: Composition of Hypercube Addresses.

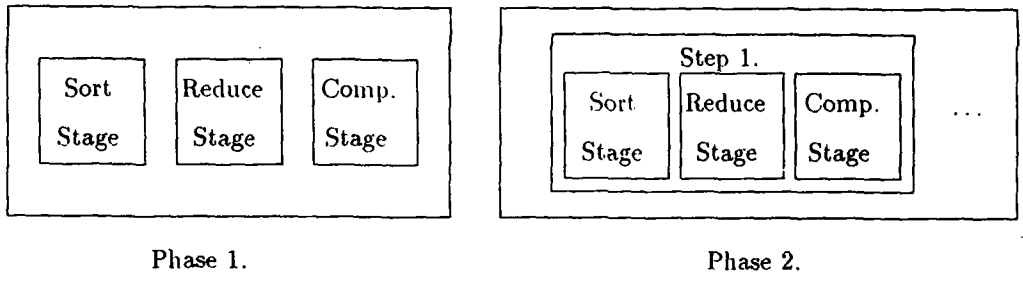

Figure 3.2: Hierarchy between Phases, Steps, and Stages.

subcubes in parallel. The hierarchy between phases, steps, and stages is illustrated in Figure 3.2.

Sort Stage. Recall that B_i is the collection of (g, v)-pairs with $g = i$. Consider a subcube, H. For convenience, we shall use $B_i(H)$ to refer to the set of (g, v)-pairs in H from a particular B_i. We shall call the set of processors holding these pairs, $proc\Big(B_i(H)\Big)$. We sort the (g, v)-pairs in H by g-value. The pairs of each $B_i(H)$ will now reside in consecutively numbered processors in H.

Reduce Stage. We next reduce each $B_i(H)$ to a single (g, v)-pair by applying the function F to the associated v-values.

Compact Stage. Finally, we route the resulting (g, v)-pairs, of which there are no more than S, into the highest-numbered processors of the subcube.

3.2.2 Auxiliary Lemmas

During the reduce stage of Phase 1, every $B_i(H)$ will be reduced to a single value in parallel in each subcube H. Before presenting the hypercube implementation of $\beta_1{}^+(N, N, S)$, we shall first establish two auxiliary lemmas that show that this reduction can be performed efficiently.

Consider a subcube, H. For a given $B_i(H)$, let us call the largest hypercube contained in $proc\Big(B_i(H)\Big)$, the *central block* of that $proc\Big(B_i(H)\Big)$ and denote it CB_i. If $B_i(H)$ has two largest hypercubes that cannot be merged, we choose the lower-numbered one.

Lemma 3.2.1 *Suppose we are given an N-node hypercube, H, holding N (g,v)-pairs sorted by g-value. Consider a given $B_i(H)$. If $2^{j+2}-1 > |proc\Big(B_i(H)\Big)| > 2^{j+1}-2$ for some j, then $|CB_i|$ must be either 2^j or 2^{j+1}.*

Proof: Clearly, $|\mathrm{CB}_i| \leq 2^{j+1}$. If we assume that $|\mathrm{CB}_i|$ is 2^h for h in the range $0 \leq h \leq j-1$, we can argue to a contradiction. Let us divide the addresses of H into left and right fields, l and r. The length of field r will be h. The address of CB_i is of the form $L\underline{R}$, where L and R are values for fields l and r, respectively. We first assume L is even. The processor at location $(L-1)0^h$ can not be in B_i, because then CB_i would have an address of the form $(L-1)\underline{R}$. The processor at location $(L+1)1^h$ can not be in B_i, because then concatenating the two hypercubes $(L)\underline{R}$ and $(L+1)\underline{R}$ would give us a CB_i of size 2^{h+1}. Thus, $|proc\Big(B_i(H)\Big)| \leq 3\cdot 2^h - 2$, which is a contradiction for all h's in the range specified.

We get an analogous contradiction if L is odd. ∎

Lemma 3.2.1 is used in Lemma 3.2.2 below.

Lemma 3.2.2 *Suppose we are given an N-node hypercube, H, holding N (g,v)-pairs sorted by g-value. We can reduce every $B_i(H)$ in H to a single value in time $O(\log N)$.*

Proof: The reduction can be performed in three steps:

Step 1. Each processor in $proc\Big(B_i(H)\Big)$ can determine if it is in CB_i, as follows. In parallel, each processor with address p checks the g-values of the two processors with addresses $p+1$ and $p-1$ to determine whether it is either the first or last processor in its $proc\Big(B_i(H)\Big)$. With two distribution-from-leaders operations[2], each processor can receive the addresses of the first and last processors in its $proc\Big(B_i(H)\Big)$. Using the results of Lemma 3.2.1, we know that if $2^{j+2}-1 > |proc\Big(B_i(H)\Big)| > 2^{j+1}-2$ for some j, then $|\mathrm{CB}_i|$ must be either 2^j or 2^{j+1}. Thus, there is at least one processor in $proc\Big(B_i(H)\Big)$ with an address having j or more trailing zeroes. Let us say that such a processor has a 0^j-*suffix*. There are at most two processors in $proc\Big(B_i(H)\Big)$ with 0^{j+1}-suffixes. (If there were three such processors, then the size of $proc\Big(B_i(H)\Big)$ would have to be at least $2^{j+2}+1$.) If there is at least one processor with a 0^{j+1}-suffix, then the lowest-numbered such processor (call it p) checks to see if the processor with address $p+2^{j+1}-1$ is a member of CB_i. If processor $p+2^{j+1}-1$ *is* a member of CB_i, then $|\mathrm{CB}_i|$ must be 2^{j+1} and p is the first processor in CB_i. If there are no processors with 0^{j+1}-suffixes, then $|\mathrm{CB}_i|$ must be 2^j and the lowest-numbered processor with a 0^j-suffix

[2]The distribution-from-leaders operation was introduced in [BH82]. A number of the processors are distinguished as *leaders*. When the distribution-from-leaders operation is performed, the leaders share their data with all the processors of higher address up to but not including the next leader.

is the first processor in CB_i . This information can be transmitted to all the processors in $proc\Big(B_i(H)\Big)$ with two additional distribution-from-leaders operations.

Step 2. There are at most $|CB_i| - 1$ processors in $proc\Big(B_i(H)\Big)$ before the first processor in CB_i. If there were $|CB_i|$ or more preceding processors, then the first $|CB_i|$ elements would constitute CB_i. There are at most $2 \cdot |CB_i| - 1$ elements after CB_i; otherwise, CB_i would be twice as large. All the processors, p_j, not in CB_i, send their triples to one of $p_{j+|CB_i|}$, $p_{j-|CB_i|}$, or $p_{j-2\cdot|CB_i|}$, depending on which of these three is an address in CB_i. The processors in these three sets can send their triples over to the appropriate processors of CB_i with three consecutive monotone routing steps.[3]

Step 3. Using simple tree operations, we can reduce each $B_i(H)$ to a single pair, in parallel. [B+86] demonstrates an embedding for the complete binary tree with $2^n - 1$ nodes in a hypercube of size 2^n. This embedding has dilation and load factor both equal to 2. The leaves of the embedded tree are mapped to the even-numbered nodes of the hypercube. By pairing the odd- and even-numbered nodes of the hypercube together, we can perform any tree operation on the hypercube with only a constant factor increase in running time.

In Step 1, each processor, p, checks the g-values of the two processors $p + 1$ and $p - 1$. This checking can be performed simply with four distribution-from-leaders operations. To check in the forward direction, we use two distribution-from-leaders operations. The odd-numbered processors are the leaders in the first such operation and the even-numbered processors are the leaders in the second. All the distribution-from-leaders operations used in this step can be performed in time $O(\log N)$ as shown in [Ull84]. The monotone routes and tree operations of the next two steps also take time $O(\log N)$.

∎

3.2.3 Hypercube Implementation - Phase 1

We start by breaking the N processors into N/S^4 hypercubes of S^4 nodes. If $S^4 > N$, we can perform the beta operation by applying Phase 1 to the entire hypercube. By convention, k is 0 for the one step of this phase; thus the hypercube addresses are of the form $u_0v_0w_0x_0$ where $|u_0| = D-4$, $|v_0| = 3$, $|w_0| = 0$, and $|x_0| = 1$. The nodes in a given subcube have addresses of the form $U_0\underline{V_0}\,\underline{X_0}$. For each subcube, H, we perform the following three stages:

Sort Stage. We use odd-even merge sort to sort the (g, v)-pairs in H by g-value in time $T_{\text{SORT}}(S^4, S^4)$. The pairs of each $B_i(H)$ will now reside in consecutively numbered processors in H.

Reduce Stage. We now invoke Lemma 3.2.2 to reduce each $B_i(H)$ to a single triple. This invocation of Lemma 3.2.2 takes time $O(\log S)$.

[3] A monotone routing step is a permutation of the items such that if there are two source nodes, σ_1 and σ_2, with two corresponding destination nodes, δ_1 and δ_2, then $\sigma_1 < \sigma_2$ implies $\delta_1 < \delta_2$. Intuitively, the items being routed maintain the same order before and after the monotonic route.

Compact Stage. At the conclusion of the reduction stage, there are S or fewer pairs in H. The pairs represent F applied to the v-values in each separate $B_i(H)$. We shall call these the final pairs. For each final pair, we compute how many other final pairs are in processors with higher addresses by means of a prefix operation. We then compact the final pairs, via a monotone route, to a subcube of size S, with address of the form $U_0\theta^3\underline{X_0}$. Both the prefix operation and the monotone route can be performed easily in time $O(\log S)$.

The compaction stage can be performed without recourse to the monotone route and prefix operation. Instead, each processor that does not contain a final pair can set its g-value to minus infinity. Sorting by g-value then compacts the final pairs into the highest-numbered processors in H, as above. The use of this sort does not increase the asymptotic time complexity, since we have already sorted once during the sort stage. Nonetheless, we choose to use the monotone route and the associated prefix operation to emphasize the fact that the initial sort is the only operation in our implementation that requires time $T_{\text{SORT}}(S^4, S^4)$. In the next section, we use this hypercube implementation as a paradigm for implementing $\beta_1{}^+(N, N, S)$ on other networks. For simplicity, we shall use sorting in the compact stages of these implementations.

3.2.4 Hypercube Implementation - Phase 2

Phase 2 is performed in $D-4$ steps. At the beginning of Step k, the remaining (g, v)-pairs will be broken into subcubes of size S^4 with addresses of the form $U_k\underline{V_k}\theta^k\underline{X_k}$. Note that with this choice of partitioning the hypercube, each subcube contains at most S^2 (g, v)-pairs. At the end of Step k, the remaining (g, v)-pairs will be compacted into subcubes of size S with addresses of the form $U_k\theta^{k+3}\underline{X_k}$. Step k $(1 \le k \le D-4)$ is performed as shown below.

Sort Stage. As in Phase 1, we consider a particular subcube, H, of size S^4. We can use the Nassimi-Sahni sort ([NS82]) to sort the at most S^2 (g, v)-pairs in H by g-value. Since the number of processors is equal to the square of the number of items to be sorted, the sorting can be performed in time $O(\log S)$.

Reduce Stage. In this stage, we shall view H as an $S^2 \times S^2$ matrix of processors, t_{ij} (see Figure 3.3). The column number shall be designated by the two least significant digits in v_k and the row number shall be designated by the remaining digit from v_k followed by the sole digit of x_k. Initially, a single row contains all the (g, v)-pairs. Using a prefix operation, we can determine which processors have the lowest addresses in their $proc\Big(B_i(H)\Big)$'s. We shall call these the *leaders*. We begin by broadcasting the contents of each initial-row processor, t_{1j}, to the column j. Next, each processor t_{jj} broadcasts to row j. Finally, in the columns of the leaders, F is applied to those v-values whose corresponding g-values match the leader's. The four tree operations of this stage can be performed in time $O(\log S)$.

Compact Stage. We consider the (g, v)-pairs that remain after the reduce stage. We first move all of these pairs to the initial row. Since there are only S groups overall, there are at most S remaining pairs in H after the reduce stage. In time $O(\log S)$, we can route these pairs into

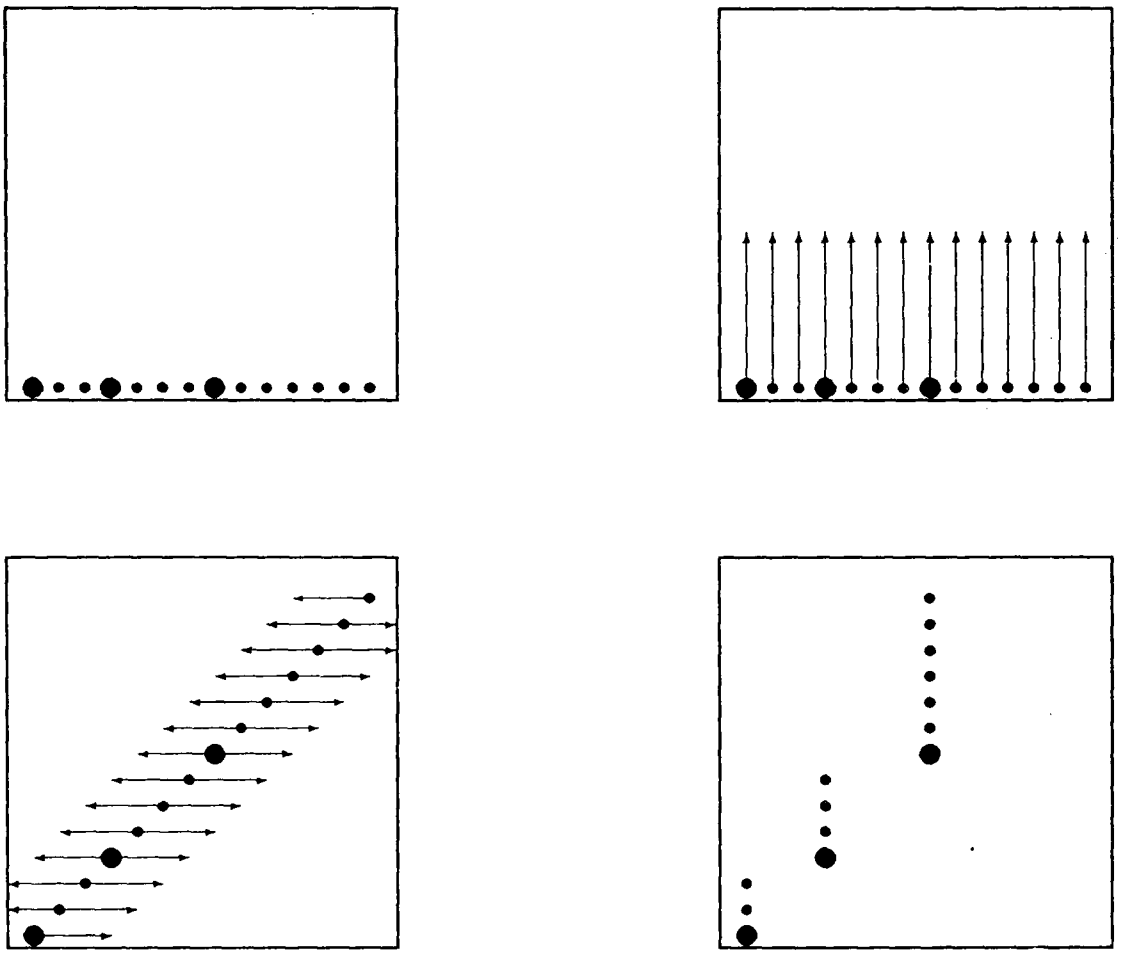

Figure 3.3: Phase 2 Reduction Stage.

the highest addresses in H with a prefix operation and a monotone route. As in Phase 1, we could use a sort to effect the compaction.

Phase 1 can be performed in time $O(T_{\text{SORT}}(S^4, S^4) + \log S)$. Since each step of Phase 2 takes time $O(\log S)$ and there are $O(\log N / \log S)$ such steps, Phase 2 takes time $O(\log N)$. The overall time used by the algorithm is thus

$$O(\log N + T_{\text{SORT}}(S^4, S^4)).$$

By Corollary A.3.1, we know that $T_{\text{SORT}}(S^4, S^4)$ is $O(T_{\text{SORT}}(S, S))$, so the overall time can be expressed as $O(\log N + T_{\text{SORT}}(S, S))$.

We stated that if $S^4 > N$, the beta operation can be performed by applying Phase 1 to the entire hypercube. This application of Phase 1 can be performed in time $O(\log N + T_{\text{SORT}}(N, N))$. This bound can be expressed as $O(\log N + T_{\text{SORT}}(S, S))$.

Additionally, using the above implementation as a paradigm, we can perform ${\beta_1}^+(N, N, S)$ on the butterfly network in time $O(\log N + T_{\text{SORT}}(S, S))$, and on the shuffle-exchange network in time $O(\log N + \log^2 S)$.

We can consider also a variant of the beta operation in which the output values, y_i, are not required to be sorted. The author knows of no algorithm for the simpler problem of computing S, given N input (g, v)-pairs, that requires time less than $O(\log N + T_{\text{SORT}}(S, S))$. Thus, it appears that this "un-sorted" variant of the beta operation is no easier to implement than the $\beta_1(N, N, S)$ operation.

3.3 Efficient Mesh-of-Trees Implementation

In this section, we show how ${\beta_1}^+(N, N, S)$ can be implemented on a mesh-of-trees network (MOT), using the hypercube implementation as a model.

We first note that ${\beta_1}^+(N, N, S)$ can be performed easily in time $O(\sqrt{N})$ on a $\sqrt{N} \times \sqrt{N}$, N-processor mesh system, even if S is not known beforehand. This upper bound is tight since there is an obvious lower bound of $\Omega(\sqrt{N})$ time, even when S is given. In the case of MOT, our results are obtained for the $\sqrt{N} \times \sqrt{N}$, $O(N)$ processor MOT system, where there is a known bound on S. As was the case with the hypercube, we shall disregard the special cases when divisions, square roots, and logarithms produce non-integral values. Although these cases present no special problems, dealing with them introduces unnecessary clutter. We shall assume that the processors are arrayed in descending row major order.

3.3.1 The General Step

The general step in Phases 2 and 3 below differs from the general step of the hypercube implementation in two ways. Firstly, the size of the sub-MOT's with which we work *grows* in each step. Recall that in each step of the hypercube algorithm, we always work with sub-hypercubes of a single size (S^4 nodes each). Secondly, each sort-reduce-compact sequence is preceded by a routing stage.

In Phase 1, we perform ${\beta_1}^+(4S, 4S, S)$ on sub-MOT's with side $\sqrt{4S}$. In Phase 2, we increase the size of the sub-MOT's considered until the number of processors is equal to the square of the number of remaining (g, v)-pairs in each sub-MOT. Finally, in Phase 3, we can increase the sub-MOT size to $\sqrt{N} \times \sqrt{N}$ quickly.

3.3.2 Auxiliary Lemmas

During the sort and compact stages of Phase 2, we shall sort the pairs in each sub-MOT. Before presenting the MOT implementation of ${\beta_1}^+(N, N, S)$, we shall first establish several auxiliary lemmas and theorems that demonstrate that this sort can be performed efficiently.

Lemma 3.3.1 *An arbitrary partial permutation routing of z elements that starts and ends on the leaves of a complete binary tree with m leaves can be performed in time $O(z + \log m)$.*

Proof: Let S_{lr} be the set of elements that needs to be routed from the left half of the tree to the right half. S_{rl}, S_{ll}, and S_{rr} are defined analogously. Using pipelining, we can easily route the elements of S_{lr} to their destinations in time $O(|S_{lr}| + \log m)$. Similarly, the elements of S_{rl} can be routed in time $O(|S_{rl}| + \log m)$. To route the elements of S_{ll}, we actually use two consecutive routings. In the first routing, the elements are routed from their source locations in the left half of the tree to the first $|S_{ll}|$ locations on the right half. The root will keep a count of the elements as they arrive and send them to the appropriate destinations. In the second routing, the elements are routed to their final destinations on the left. Both routings take time $O(|S_{ll}| + \log m)$. Similarly, the elements in S_{rr} can be routed in time $O(|S_{rr}| + \log m)$. Since $z = |S_{rl}| + |S_{lr}| + |S_{ll}| + |S_{rr}|$, the overall routing can be accomplished in time $O(z + \log m)$. ∎

Lemma 3.3.2 *Suppose we are given a MOT of side m, with all elements contained in the first z rows. In time $O(z + \log m)$, we can achieve any permutation in which the final destinations of the elements are also within the first z rows.*

Proof: Let $R_{i,j}$ be the row of the destination of the element that starts in row i, column j; similarly, $C_{i,j}$ is the column of the destination. We apply Lemma 3.3.1 three times. The lemma is applied first to the columns, then to the rows and finally to the columns of the MOT so that each element from (i, j) follows the permutations: $(i, j) \rightarrow (i + j, j) \rightarrow (i + j, C_{i,j}) \rightarrow (R_{i,j}, C_{i,j})$, where all additions are mod m. Note that the first application of Lemma 3.3.1 staggers the elements so that no row contains more than z elements (see Figure 3.4). Each of these three permutation operations can be performed in $O(z + \log m)$ time, yielding the desired result. ∎

Theorem 3.3.1 *Consider a MOT with side m. Assume that it is divided vertically into two halves. Assume further that the first z ($1 \leq z \leq m$) rows on the left side contain the sorted list, A, and the first z rows on the right side contain the sorted list, B. We can merge these two lists into one list located in the first z rows, in time $T(z, m) = O(z + \log m \log z)$.*

Proof: The result is achieved by using odd-even merge (see Figure 3.5).

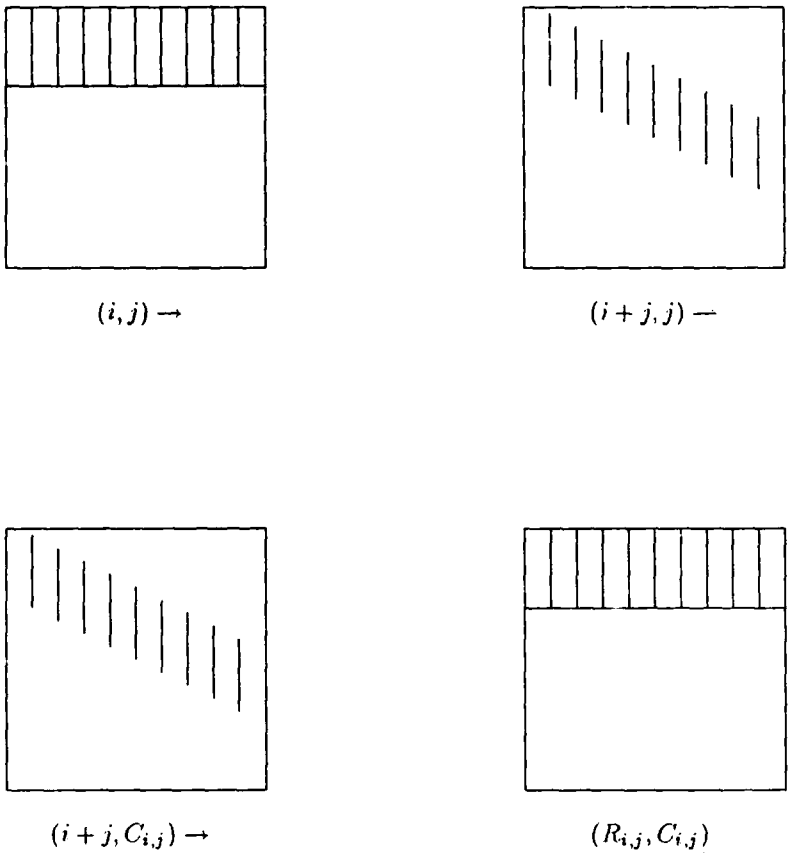

Figure 3.4: Permutations of Lemma 3.3.2.

Step 1. We start by separating the odd-position elements of A (A_{odd}) from the even-position elements (A_{even}) so that A_{odd} occupies the first $z/2$ rows and A_{even} occupies the $z/2$ rows starting at row $m/2$. In Figure 3.5, we shall abbreviate the subscripts, (e.g., A_{odd} will be denoted as A_o.)

Simultaneously, we separate the B's.

Step 2. We next exchange the positions of A_{odd} and B_{even}.

Step 3. We now want to merge lists that are stacked vertically. Consider the $m/2 \times m/2$ square in the upper left. We separate out the even-position elements of B_{even} ($B_{even\text{-}even}$) and the odd-position elements of B_{even} ($B_{even\text{-}odd}$). The elements of $B_{even\text{-}even}$ are moved to the left half of the space previously occupied by B_{even} and the odd-position elements are moved to the right. At the same time, we perform the analogous separations on the elements of B_{odd}, A_{even} and A_{odd}.

Step 4. We next exchange $B_{even\text{-}even}$ and $A_{even\text{-}odd}$. Simultaneously, we also exchange $B_{odd\text{-}even}$ and $A_{odd\text{-}odd}$.

Step 5. We now have 4 sub-MOT's with side $m/2$ and $z/2$ rows. We can recursively perform merge on the lists in these sub-MOT's in time $T(z/2, m/2)$ to yield the four lists $AB_{even\text{-}even}$, $AB_{even\text{-}odd}$, $AB_{odd\text{-}even}$, and $AB_{odd\text{-}odd}$.

Step 6. We then interleave $AB_{even\text{-}even}$ with $AB_{even\text{-}odd}$. Using only swaps of adjacent list elements, we can produce the sorted list AB_{even}. We simultaneously perform the same interleave-swap operation on $AB_{odd\text{-}even}$ and $AB_{odd\text{-}odd}$, yielding the sorted list AB_{odd}. Lastly, we interleave AB_{even} with AB_{odd}, and perform any necessary value swapping.

Basis. If $z = 1$ then we can sort in time $O(\log m)$.

Induction step. Let $z = z_0$. The permutations in the above steps can all be performed in time $c(z + \log m)$ for some constant c by invoking Lemma 3.3.2 and using simple pipelining. Thus, $T(z_0, m) \leq T(z_0/2, m/2) + c'(z_0 + \log m)$ for some constant c' and $T(z, m) = O(z + \log m \log z)$.

∎

Theorem 3.3.2 *Consider a MOT with side m. Assume that there are $O(mz)$ elements in the first z rows. We can sort these elements with the results ending up in the first z rows, in time $T(z, m) = O(z + \log m \log^2 z)$.*

Proof: We use a merge sort. We first divide the MOT into four sub-MOT's of side $m/2$. We can distribute the elements into the first $z/2$ rows of each sub-MOT with simple pipelining. We then recursively sort the elements in each sub-MOT in time $T(z/2, m/2)$. We next merge the four sorted lists together, using the methods outlined in Theorem 3.3.1. Hence, $T(z, m) = T(z/2, m/2) + c(z + \log m \log z)$ for some constant c. Solving this recurrence yields the time bound claimed above. ∎

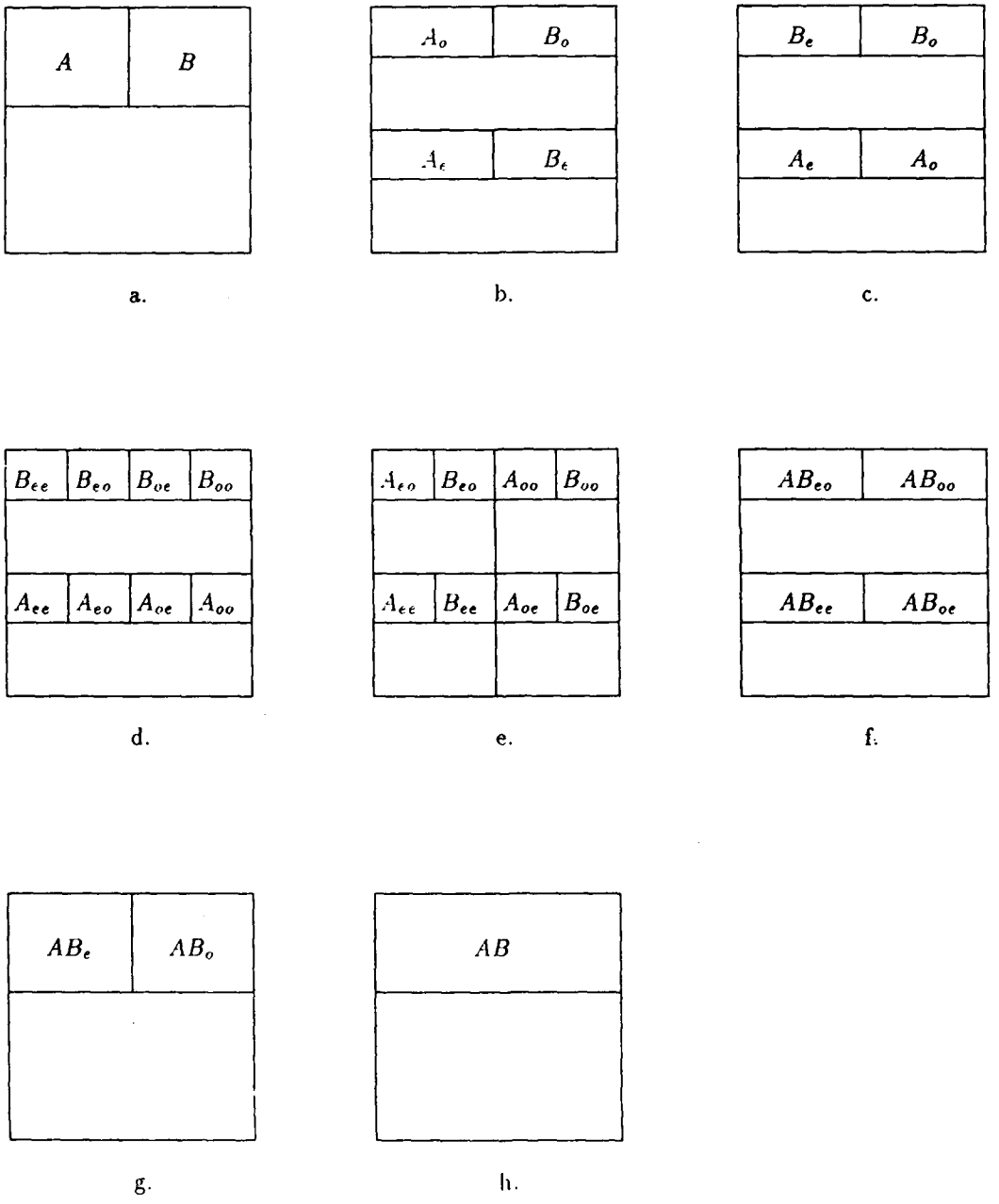

Figure 3.5: Steps of the Odd-Even Merge.

3.3.3 MOT Implementation - Phase 1

We first break the N processors into $N/4S$ sub-MOT's with side $\sqrt{4S}$. For each sub-MOT, M, we perform $\beta_1{}^+(4S, 4S, S)$ by applying the following three stages.

Sort Stage. We first sort the pairs in each sub-MOT by g-value. This sort can be accomplished in time $O(\sqrt{S})$ by invoking Theorem 3.3.2.

Reduce Stage. Let $B_i(M)$ refer to the set of (g, v)-pairs in M from a particular B_i. Using tree operations, we can combine the (g, v)-pairs from each $B_i(M)$ to produce one representative (g, v)-pair in time $O(\log S)$.

Compact Stage. Each processor not holding one of the resultant pairs sets its g-value to minus infinity. We then sort again on g-value, so that the remaining (g, v)-pairs are compacted in the first S spaces (in the row-major sense).

3.3.4 MOT Implementation - Phase 2

Phase 2 is performed in $(3s/2) - 1$ steps. At the beginning of Step k, the MOT is divided into $N/(4^k S)$ sub-MOT's with sides of length $\sqrt{4^k S}$. The first $\left\lceil \sqrt{S/4^k} \right\rceil$ rows of each such sub-MOT contain the S or fewer different (g, v)-pairs, compacted to the left. For convenience, these initial rows of the sub-MOT shall henceforth be called the non-trivial-part (NTP). Step k $(1 \leq k \leq 3s/2 - 1)$ is performed with the following sequence of stages.

Route Stage. We start the routing stage by conceptually clumping 4 contiguous sub-MOT's into a single square sub-MOT with twice the side length. We first shift up the NTP's of the two lower sub-MOT's so that they are contiguous to the NTP's of the upper sub-MOT's. This results in a larger sub-MOT with side $\sqrt{4^{k+1}S}$ having a NTP occupying the first $2\left\lceil \sqrt{S/4^k} \right\rceil$ rows. This routing can be done with simple pipelining in time $O(\left\lceil \sqrt{S/4^k} \right\rceil)$.

Sort Stage. We then sort this new NTP by invoking Theorem 3.3.2. The sorting of Step k can be performed in time

$$O\left(\left\lceil \sqrt{S/4^k} \right\rceil + \log \sqrt{4^{k+1}S} \log^2 \left\lceil \sqrt{S/4^k} \right\rceil\right).$$

Reduce Stage. For each group number, there are up to four different (g, v)-pairs. We can combine these pairs into one (g, v)-pair in time $O(\log \sqrt{4^{k+1}S})$.

Compact Stage. As in Phase 1, we compact the NTP into the first S spaces (in the row-major sense) by means of a sort.

Summing the time requirements of each step, we see that Phase 2 can be performed in time

$$O\left(\sum_{k=1}^{3s/2} 2\left\lceil \sqrt{S/4^k} \right\rceil + \log \sqrt{4^{k+1}S} \log^2 \left\lceil \sqrt{S/4^k} \right\rceil\right)$$

$$= O(\sqrt{S} + \log^4 S) = O(\sqrt{S}).$$

At the end of this phase, we have N/S^4 sub-MOT's of side S^2, each with no more than S non-trivial (g, v)-pairs.

3.3.5 MOT Implementation - Phase 3

Phase 3 is performed in $\log(n/2s - 1)$ steps. In Step k, we shall increase the side of the sub-MOT from $S^{2^{k-1}+1}$ to S^{2^k+1}. The last step will leave us a single MOT with side $\sqrt{N}$. Note that in each sub-MOT, the number of processors will always be equal to the square of the number of non-trivial (g, v)-pairs. Step k $(1 \leq k \leq \log(n/2s - 1))$ is performed as shown below.

Route Stage. At the beginning of Step k, we have sub-MOT's of side $S^{2^{k-1}+1}$, each with no more than S non-trivial (g, v)- pairs. We shall conceptually clump S^{2^k} of these sub-MOT's into sub-MOT's of side S^{2^k+1}. We shall view each such sub-MOT of side S^{2^k+1} as being composed of $S^{2^{k-1}}$ columns of width $S^{2^{k-1}+1}$. To route, we move the NTP's from the sub-MOT's in each column into the diagonal processors of that column as a preliminary to sorting. We follow the lead of [Ull84] in calling the processor at location (i, i), the ith controller. The routing can be done with the tree connections in time $O(\log S^{2^k+1})$.

Sort Stage. For each clump, we have a sub-MOT of $S^{2^{k+1}+2}$ processors and no more than S^{2^k+1} non-trivial (g, v)-pairs. Since the number of processors is at least equal to the square of the number of pairs, the sorting can be performed in time $O(\log S^{2^k+1})$ using the standard MOT sorting algorithm [Ull84].

Reduce Stage. The reduce stage closely resembles the standard MOT sorting algorithm. First, every controller checks to see if the group number it contains is the leftmost such group number. As with the hypercube algorithm, we shall call such processors, leaders. Next, each controller broadcasts its value to its entire row and column. Finally, in the columns of the leaders, F is applied to the v-values whose corresponding g-values match that of the column's leader. These tree operations can be performed in time $O(\log S^{2^k+1})$.

Compact Stage. Another sort will then compact the values. As in the compact stage of Phase 2, it is assumed that non-leader controllers have g-values equal to minus infinity. The sort takes the same time as in the sort stage above.

The third phase, therefore, takes time $O(\sum_{k=1}^{\log(n/2s)} \log S^{2^k+1}) = O(n)$.

The overall time used by the MOT algorithm is thus

$$O(\log N + \sqrt{S}).$$

Note that for both the mesh and the MOT, $T_{\text{SORT}}(S, S) = \Theta(\sqrt{S})$. Thus, the overall time required can be expressed as $O(\log N + T_{\text{SORT}}(S, S))$.

3.4 Determining the Output Size

The running times of the algorithms given above are functions of both the input size, N, and the output size, S. These algorithms assume that S is known. Thus, the question arises, what do we do if we do not know the output size, S?

For a large class of problems, to which beta operations appear to belong, the problem of determining the output size, S, is essentially as complex as the problem of computing the output, given the output size. While it would be possible to develop separate algorithms to determine the output size, we shall exhibit below a general scheme that enables one to determine S *and* solve the problem in time proportional to that required for solving the problem alone, ***given*** S.

3.4.1 Iterative Guessing

We shall create an algorithm, call it Algorithm B, for solving 'problem $\mathcal{P}$ not given output size S.' We shall employ a strategy of iterative guessing that consists of two components. Firstly, we require an algorithm, call it Algorithm A, for 'problem $\mathcal{P}$ given output size, S.' Secondly, we require an increasing sequence of guesses for the output size, $\Gamma = (g_1, g_2, \ldots)$. In Algorithm B, we shall sequentially test the guesses in Γ on Algorithm A. That is, we shall first run Algorithm A with output size equal to g_1. If this first guess is too small, we shall run Algorithm A with output size equal to g_2, then g_3, etc... until Algorithm A finally succeeds.

This guessing strategy will be efficient if Algorithm A and sequence Γ possess certain properties. Algorithm A must satisfy the following four conditions:

(i) The running time is $t(N, Q)$, where Q is a given bound on the output size.

(ii) If $Q \geq S$, the algorithm works correctly and produces the appropriate output of size S.

(iii) If $Q < S$, the algorithm can detect the error (within time $t(N, Q)$).

(iv) $t(N, Q)$ is monotonically increasing in Q.

Let the minimum output size possible for any input be $S_{\min}$. Sequence Γ must satisfy the following two conditions:

(a) $g_0 = S_{\min}$

(b) $\exists\, c_1, c_2$ s.t. $1 < c_1 \leq c_2$ and $c_1 t(N, g_{i-1}) \leq t(N, g_i) \leq c_2 t(N, g_{i-1})$

The constants, c_1 and c_2, are independent of i, but can be chosen in a fashion that depends on the elements of Γ. The efficiency of the iterative guessing strategy is demonstrated in the following theorem.

Theorem 3.4.1 *Assume that we are given an algorithm for 'problem $\mathcal{P}$ given S' that satisfies conditions (i)-(iv). Then, if we are given an associated guess sequence, Γ, satisfying conditions (a)-(b), we can create an algorithm to solve $\mathcal{P}$ not given S, in time $\tilde{t}(N, S)$ where $\tilde{t}(N, S) = \Theta(t(N, S))$.*

Proof: The given algorithm for '$\mathcal{P}$ given S' is our 'Algorithm A'. To solve the problem '$\mathcal{P}$ not given S', we run 'Algorithm B'. Let g_f be the guess for Q on which the Algorithm A finally succeeds.

From the properties of Algorithm A, it follows that $g_{f-1} < S \leq g_f$. Hence,

$$\begin{aligned} t(N, g_f) &\leq c_2 t(N, g_{f-1}) \\ &\leq c_2 t(N, S). \end{aligned}$$

Also,

$$\begin{aligned} c_1 t(N, g_{i-1}) &\leq t(N, g_i) \\ \sum_{i=2}^{f} c_1 t(N, g_{i-1}) &\leq \sum_{i=2}^{f} t(N, g_i) \\ t(N, g_1) + \sum_{i=1}^{f-1} (c_1 - 1) t(N, g_i) &\leq t(N, g_f) \\ \sum_{i=1}^{f-1} t(N, g_i) &\leq \frac{1}{c_1 - 1} t(N, g_f) \\ \sum_{i=1}^{f} t(N, g_i) &\leq \frac{c_1}{c_1 - 1} t(N, g_f). \end{aligned}$$

From the definition of our algorithm for '$\mathcal{P}$ not given S',

$$\begin{aligned} \tilde{t}(N, S) &\leq \sum_{i=1}^{f} t(N, g_i) \\ &\leq \frac{c_1}{c_1 - 1} t(N, g_f) \\ &\leq \frac{c_1 c_2}{c_1 - 1} t(N, S). \end{aligned}$$

Since it is trivially true that

$$\tilde{t}(N, S) \geq t(N, S)$$

it follows that $\tilde{t}(N, S) = \Theta(t(N, S))$. Note that the optimal choice of $c_1 = c_2 = 2$ yields a factor of 4 slowdown in the worst case. ∎

3.4.2 Application of Method

Lemma 3.4.1 *Let $t(N, Q) = c(\log N + F(Q))$ where $F(Q)$ is a monotonically increasing function and c is a constant. The guess sequence*

$$g_0 = 1, \; g_i = min\{x \mid F(x) = ((2^i - 1) \log N)\}, \; i > 0$$

satisfies conditions (a)-(b) with $c_1 = c_2 = 2$.

The algorithm described in Section 3.2 satisfies conditions (i)-(iv) and the associated running time is of the form described in Lemma 3.4.1. Using the knowledge that $T_{\text{SORT}}(S^4, S^4)$ is $d \cdot T_{\text{SORT}}(S, S)$ for some constant d (as shown in Appendix A), we conclude that the corollary below follows from Lemma 3.4.1 and Theorem 3.4.1.

Corollary 3.4.1 *$\beta_1{}^+(N, N, S)$ can be implemented on a hypercube in time*

$$O(\log N + T_{\text{SORT}}(S, S)),$$

without prior knowledge of S.

The algorithm described in Section 3.3 also satisfies conditions (i)-(iv) and the associated running time is of the form described in Lemma 3.4.1. The corollary below, therefore, follows from Lemma 3.4.1 and Theorem 3.4.1.

Corollary 3.4.2 *$\beta_1{}^+(N, N, S)$ can be implemented on a MOT in time*

$$O(\log N + \sqrt{S}),$$

without prior knowledge of S.

Recall that since $T_{\text{SORT}}(S, S) = \Theta(\sqrt{S})$ for the MOT, the time for the MOT implementation of $\beta_1{}^+(N, N, S)$ can be expressed as $O(\log N + T_{\text{SORT}}(S, S))$.

3.5 Lower Bounds

In this section, we shall prove some lower bounds, given our formulation of the beta operation, and relate them to the areas and times associated with the algorithms and architectures discussed above. Note that while in the other sections of this chapter we use the word model of computation, here we use the bit model of computation.

The input consists of N pairs of numbers, $(g_0, v_0), (g_1, v_1), \ldots, (g_{N-1}, v_{N-1})$. We shall demonstrate the lower bound for the case where these numbers are constrained to be in the range 0 to $N-1$. Recall from Section 2.1 that $l_i = \{v_j \mid v_j \in B_i\}$. For all i such that $|l_i| > 0$, we output (i, y_i) sorted by i, where y_i is the F-reduction of l_i. Recall also that $G = \{i \mid |B_i| > 0\}$. Let w_0 be the smallest member of G; similarly, let w_i $(i < |G|)$ be the $(i+1)$-th smallest member of G. Define $z_i = y_{w_i}$; that is, z_i is the y-value of the $(i+1)$-th output. We shall refer to the j-th bit of z_i as $z_{i,j}$. (Similarly for the g's.)

3.5.1 A Lower Bound on Area

The structure of the following lower bound proof follows closely that of the proof that appears in Example 2.3 in [Ull84]. We begin by introducing the following lemma.

Lemma 3.5.1 *In a when- and where-determinate circuit that performs the beta operation correctly for any S, none of the output bits $z_{i,j}$ (for $i < N-1$) can be output before all of the input bits $g_{i,j}$ (for $j > 0$) have been read.*

Proof: Assume, to the contrary, that $z_{p,q}$ ($p < N-1$) is output before $g_{w,x}$ ($x > 0$) is read. We construct two possible inputs by fixing every g and v, except g_w, as follows:

- Choose a $t \neq w$. Set $g_t = p+1$; $v_t = 2^q$.
- Set all other $v_i = 0$.
- For all i (other than $i = t$ and $i = w$), choose a value of g_i (different from p and $p+1$) in such a way that $\forall j$, $0 \leq j < p$, $\exists i$ such that $g_i = j$.

The two possible inputs yielded by setting either $g_w = p$ or $g_w = p \oplus 2^x$ produce different values for the bit $z_{p,q}$. Yet $z_{p,q}$ is output before $g_{w,x}$ is read – contradiction. ∎

Theorem 3.5.1 *Any when- and where-determinate circuit that can perform the beta operation correctly for any S must have $A = \Omega(N \log N)$.*

Proof: (This proof assumes N is even. The proof for N odd is similar.) We shall construct a family of inputs of size $(N/2)!$, each with different outputs. For all i, fix $v_i = i$. For $i \geq N/2$, fix $g_i = 2i - N + 1$. Now, we allow the remaining inputs, $g_0, \ldots, g_{N/2-1}$ to be any permutation of the even numbers less than N.

Focus on the time just before the first $z_{i,j}$ (for $i < N-1$) is output. The circuit has already read all of the bits that differ between the elements of our family of inputs. Hence, all inputs read subsequently will be the same for any element of our family. Since the circuit must still produce $(N/2)!$ different outputs, it must have at least $(N/2)!$ states, and hence, at least $\log((N/2)!) = \Omega(N \log N)$ bits and area. ∎

3.5.2 An AT^2 Lower Bound

The structure of this lower bound proof closely follows that of the lower bound proof in Section 5 of [Hoc85].

Theorem 3.5.2 *Any when- and where-determinate circuit that can perform beta operations for any S requires $AT^2 = \Omega(NS \log N)$, where N is the number of input pairs.*

Proof: If there is an input pad that reads $\Omega(S^{1/2})$ bits of the v_i's, then $T = \Omega(S^{1/2})$. This condition, coupled with Theorem 3.5.1, implies our AT^2 bound.

If no pad reads $\Omega(S^{1/2})$ bits, then we may partition our circuit into a set of blocks B with $|B| = \Theta(\frac{N \log N}{S})$ so that

- each block in B has area $O(\frac{AS}{N \log N})$ and perimeter $O(\sqrt{\frac{AS}{N \log N}})$;

- each block in B reads at most $O(S)$ bits of the v_i's.

Such a collection of blocks is obtained by first cutting the circuit into $\Theta(\frac{N \log N}{S})$ blocks each of perimeter $O(\sqrt{\frac{AS}{N \log N}})$ (Lemma 2.3 of [Ull84]). Then another $\Theta(\frac{N \log N}{S})$ cuts suffice to ensure that each resulting block reads at most $I_{\max} = O(S)$ bits of the v_i's.

Two statements are true of the above set of blocks:

- At least half of these blocks produce less than twice the average number, $\Theta(\frac{S^2}{N})$, of the output z_i bits for $i < S$.
- Let I_{ave} be defined as $N \log N/|B| = \Theta(S)$. More than half of the blocks read at least $2I_{\text{ave}} - I_{\max}$ input bits.

Note that we can stay within our block cutting rules and still make our blocks small enough so that $2I_{\text{ave}} - I_{\max} = \Omega(S)$. Thus, there is some block, $b \in B$, that:

- reads at least $l_1 = \Omega(S)$ input bits from l_2 of the v_i's (assume without loss of generality that these are from $v_0, \ldots, v_{l_2-1}$);
- has perimeter $O(\sqrt{\frac{AS}{N \log N}})$;
- produces $l_3 = O(\frac{S^2}{N})$ of the z_i output bits with $i < S$.

We may then construct a fooling set as follows. Let

$$V = \{(i,j) \mid b \text{ reads in bit } j \text{ of } v_i\}$$

$$Z = \{i \mid b \text{ outputs a bit of } z_i, \text{ and } i < S\}.$$

For each $1 \le i \le l_2$, choose a distinct α_i, such that $\alpha_i \notin Z$ and $0 \le \alpha_i < S$. (Note that we can stay within our block-cutting rules and still make our blocks small enough so that $l_2 + l_3 < S - 1$.) Let $c = S - l_2$. Choose $\beta_1, \ldots, \beta_c$ (each $\beta < S$) to be distinct from each other and the α_i's. Choose the g_i's as follows

$$\begin{aligned} g_i &= \alpha_i, && \text{for } i = 1 \text{ to } l_2 \\ g_{i+l_2} &= \beta_i, && \text{for } i = 1 \text{ to } c \\ g_i &= \beta_1, && \text{for all other } i. \end{aligned}$$

And the v_i's as

$$\begin{aligned} \text{bit } j \text{ of } v_i &= 0 \text{ or } 1, && \text{for } (i,j) \in V \\ \text{bit } j \text{ of } v_i &= 0, && \text{for } (i,j) \notin V \end{aligned}$$

Each of our 2^{l_1} choices for the input yields a different configuration of the output bits outside of b. As $l_1 = \Omega(S)$, the fooling set is of size $2^{\Omega(S)}$. This yields the bound $\sqrt{\frac{AS}{N \log N}} T = \Omega(S)$; that is, $AT^2 = \Omega(NS \log N)$. ∎

In Section 3.3, we showed that the problem can be solved in time $O(\sqrt{S} + \log N)$ on a MOT. The resulting AT^2 bound of $N \log^2 N(S + \log^2 N)$ is within a few $\log N$ factors of the lower bound of $AT^2 = \Omega(NS \log N)$ shown above, even after accounting for the word model vs. bit model distinction.

Chapter 4

Generalized Beta Operations

4.1 Overview

In Chapter 3, we considered the efficient implementation of ${\beta_1}^+(P, P, S)$. In this chapter, we present efficient implementations of ${\beta_1}^+(N, P, S)$ for all possible relationships among S,N, and P. We shall concentrate on the hypercube network.

We first note that the number of groups represented in the pairs can be no larger than the number of pairs (i.e., $S \leq N$). Thus, there are three possible relationships among the variables S,N, and P. We shall denote these relationships as:

Case 1: $S \leq N \leq P$

Case 2: $S \leq P \leq N$

Case 3: $P \leq S \leq N$

In Sections 4.2, 4.3, and 4.4, we shall deal with the question of how to implement ${\beta_1}^+(N, P, S)$ efficiently in these three cases, when S is known. The running times of these implementations are all expressed in terms of $T_{\text{SORT}}(N, P)$. In Section 4.5, we provide actual time bounds for the three cases. In Section 4.6, we shall generalize the result of Section 3.4, demonstrating that the same time bounds can be achieved without knowing S beforehand.

We shall follow the convention that the S output pairs end up sorted by g-value, $\lceil S/P \rceil$ per processor, in the first E processors, where $E = \text{Minimum}(S, P)$. In what follows, we shall let N/P be called R. We shall also call $\log R$, r.

4.2 Case 1: $S \leq N \leq P$

Theorem 4.2.1 *Let $Z = \text{Minimum}(S^3, N)$. If we are given N (g, v)-pairs, distributed one or fewer per processor, in a P-processor hypercube, then ${\beta_1}^+(N, P, S)$ can be performed in time*

$$O\left(\log P + T_{\text{SORT}}(Z, \frac{Z}{R})\right)$$

when $P \geq N$.

Proof: This result is obtained easily by modifying the Section 3.2 implementation of ${\beta_1}^+(P,P,S)$. We consider two possible cases.

[$Z = S^3$] Recall that aside from the sort stage of Phase 1, the Section 3.2 implementation of ${\beta_1}^+(P,P,S)$ requires time $O(\log P)$. Since $Z = S^3$ (i.e., $S^3 \leq N$), it suffices to show that the sort stage of Phase 1 of the Section 3.2 algorithm can be performed in time

$$O(\log P + T_{\mathrm{SORT}}(S^3, \frac{S^3}{R})).$$

We start by redistributing the pairs. With the use of a simple prefix operation, each pair finds the number of pairs with higher-numbered addresses. Using a monotone route, we can route the pairs so that each set of S^3 pairs is contained in a subcube with S^3/R processors. The pairs in each subcube can then be sorted in parallel in time $T_{\mathrm{SORT}}(S^3, S^3/R)$. Using one more prefix operation and monotone route, we can compact the pairs in the N highest-address processors. The monotone routes and prefix operations can be performed in time $O(\log P)$.

[$Z = N$] We start by sorting all the pairs in time $T_{\mathrm{SORT}}(N,P)$. ${\beta_1}^+(N,P,S)$ can then be completed using the reduce and compact stages of Phase 1 of the Section 3.2 algorithm applied to the entire hypercube. Since $Z = N$ (i.e., $S^3 > N$), the total time required is $O(\log P + T_{\mathrm{SORT}}(Z, Z/R))$.

∎

4.3 Case 2: $S \leq P \leq N$

In this section, we deal with the question of how to implement ${\beta_1}^+(N,P,S)$ efficiently in the case where $S \leq P \leq N$. We begin by introducing two simple auxiliary lemmas:

Lemma 4.3.1 *We can perform* ${\beta_1}^+(N,1,S)$ *in time* $N \log S$.

Proof: We shall keep a balanced search tree of (g,v)-pairs representing g-values encountered so far. We visit the N pairs in turn. For each unvisited pair, (g_i, v_i), we first check whether group g_i is represented in the tree; if so, we combine the value v_i with the associated v-value. If g_i is not in the tree, then we insert the pair (g_i, v_i) into the tree. The time required to update the tree in either case is $O(\log S)$. The time to process all N pairs is thus $O(N \log S)$. ∎

Lemma 4.3.2 *Suppose we are given a P-node hypercube, with each processor holding two (g,v)-pairs. We can perform* ${\beta_1}^+(2P,P,S)$ *in time*

$$O(\log P + T_{\mathrm{SORT}}(S,S))$$

when $S \leq P$.

Proof: We start by performing $\beta_1{}^+(P, P, S)$ once on the first pairs of each processor and then again on the second pairs. Let us call the pairs remaining from the first $\beta_1{}^+(S)$ operation, A. Let us call the pairs held by the $|A|/2$ highest-numbered processors, A_H (respectively A_L for the other pairs of A). The output pairs resulting from the second $\beta_1{}^+(S)$ operation will be termed B_H and B_L, analogously.

Consider the following series of operations on A_H and B_H. A_H and B_H are first merged by g-value. Each processor, p, holding a pair from A_H then checks to see if either neighboring processor is holding a pair from B_H with the same g-value. If there is such a neighboring processor, q, the v-values of the two pairs are combined and placed in the v-slot of p's pair. The (g, v)-pair of q is then deleted. Finally, the initial merge is reversed on the pairs of A_H and the remaining pairs of B_H. As a result of these operations, no g-value appears in a pair of both A_H and B_H. Note that this series of operations can be performed easily in time $O(\log P)$.

The above operations are then repeated on A_H and B_L, A_L and B_L, and A_L and B_H. Following these operations, A_H, A_L, B_H, and B_L are disjoint with respect to their g-values. We can then complete $\beta_1{}^+(2P, P, S)$ by performing $\beta_1{}^-(P, P, S)$ on B followed by $\beta_1{}^+(P, P, S)$ on all the remaining pairs. Since $\beta_1(S)$ is performed a constant number of times, the overall time required is

$$O(\log P + T_{\text{SORT}}(S, S)).$$

∎

There are two possible subcases: either $R > S$ or $R \leq S$. We shall consider these two possibilities in turn.

4.3.1 $R > S$ $(N > PS)$

Theorem 4.3.1 *If we are given N pairs distributed evenly at the P processors of a hypercube and $R > S$, then we can perform $\beta_1{}^+(N, P, S)$ in time*

$$O(R \log S + \log P + T_{\text{SORT}}(S, S))$$

when $S \leq P \leq N$.

Proof:

[Phase 1] We start by invoking Lemma 4.3.1 to perform $\beta_1{}^+(R, 1, S)$ in parallel at each processor. $\beta_1{}^+(R, 1, S)$ takes time $O(R \log S)$. There are now at most S remaining pairs at each processor. Note that these remaining (g, v)-pairs are now in sorted order.

[Phase 2] We now group the processors into subcubes of size S. Consider a particular such subcube, H. We repeat the following sequence of three steps on all the subcubes in parallel until there are no more (g, v)-pairs to be considered:

1. Find the minimum g-value among the remaining (g, v)-pairs in H.

2. Find the result of applying F to the list of v-values of all the (g, v)-pairs with this g-value and remove these pairs.
3. Send the resulting pair to the highest-numbered processor in H not yet holding a representative pair.

Since the initial $\beta_1{}^+(R, 1, S)$ leaves the pairs sorted at each processor, each step in Phase 2 can be accomplished with a simple tree operation. Thus, the entire phase takes time $O(S \log S)$. After this phase, there is at most one (g, v)-pair at each processor.

[Phase 3] We then perform the Section 3.2 algorithm for computing $\beta_1{}^+(P, P, S)$. This algorithm requires time $O(\log P + T_{\text{SORT}}(S, S))$.

By adding the time requirements for each phase, we can see that the overall time required is $O(R \log S + \log P + T_{\text{SORT}}(S, S))$. ∎

4.3.2 $R \leq S$ $(N \leq PS)$

Theorem 4.3.2 *Let $Z = Minimum(S^2, N)$. If we are given N pairs distributed evenly at the P processors of a hypercube and $R \leq S$, then we can perform $\beta_1{}^+(N, P, S)$ in time*

$$O(R \log S + \log P + T_{\text{SORT}}(Z, Z/R))$$

when $S \leq P \leq N$.

Proof:

We consider two possible cases.

[$Z = S^2$] We start by grouping the processors into subcubes of size S^2/R. Consider a particular such subcube, H. We shall perform the following three phases on each subcube in parallel:

[Phase 1] Since there are R pairs at each processor, there are S^2 pairs in H. The pairs of H can be sorted in time $T_{\text{SORT}}(S^2, S^2/R)$.

[Phase 2] Each processor in H performs $\beta_1{}^+(R, 1, S)$ on the pairs it contains. Since the pairs are already sorted, this operation can be performed in time linear in the number of pairs. Reasoning as below, we can conclude that after this beta operation is performed, there are no more than $2S^2/R$ pairs remaining in H.

Let ξ_{g_i} be one fewer than the number of contiguous processors in H that contain pairs from g_i, after the sort of Phase 1. After the initial $\beta_1{}^+(R, 1, S)$, the number of remaining pairs representing group g_i is $\xi_{g_i} + 1$.

$$\sum_{g_i \in G} \xi_{g_i} \leq S^2/R = |H|$$

Hence,

$$\sum_{g_i \in G} (\xi_{g_i} + 1) \leq S^2/R + S \leq 2S^2/R$$

[Phase 3] After Phase 2, there are are no more than $2S^2/R$ pairs left in H, and no more than R pairs left in any given processor. Ullman ([Ull]) showed that in time $O(R \log S)$, the pairs of H can be redistributed so that there are no more than two pairs per processor.

To complete the implementation, we now perform $\beta_1{}^+(2P, P, S)$ on the whole hypercube by invoking Lemma 4.3.2. The total required time, expressed in terms of Z, simplifies to

$$O(R \log S + \log P + T_{\text{SORT}}(Z, Z/R)).$$

[$Z = N$] If $Z = N$ (i.e., $S^2 > N$), $\beta_1{}^+(N, P, S)$ can be performed by applying the above three phases to the whole hypercube. Phase 1 requires time $T_{\text{SORT}}(N, P)$. Phase 2 requires time $O(R)$, as before. After Phase 2, there are at most $P + S \leq 2P$ remaining pairs and no more than R pairs left in any given processor. The redistribution of Phase 3 requires time $O(R \log P)$. Since $S^2 > N \geq P$, $\log P = O(\log S)$, so Phase 3 requires time $O(R \log S)$. The final $\beta_1{}^+(2P, P, S)$ requires time $O(\log P + T_{\text{SORT}}(S, S))$, as before. The total required time, expressed in terms of Z, simplifies to

$$O(R \log S + T_{\text{SORT}}(Z, Z/R)).$$

∎

4.4 Case 3: $P \leq S \leq N$

In this section, we deal with the question of how to implement $\beta_1{}^+(N, P, S)$ efficiently in the case where $P \leq S \leq N$. As in Section 4.3, there are two possibilities: $R > S$ and $R \leq S$. We shall consider them in turn.

4.4.1 $R > S$ $(N > PS)$

Theorem 4.4.1 *If we are given N pairs distributed evenly at the P processors of a hypercube and $R > S$, then we can perform $\beta_1{}^+(N, P, S)$ in time $O(R \log S)$ when $P \leq S \leq N$.*

Proof: The algorithm for this subcase mirrors the algorithm of Section 4.3.1.

[Phase 1] We start by invoking Lemma 4.3.1 to perform $\beta_1{}^+(R, 1, S)$ separately at each processor. This operation takes time $O(R \log S)$ and reduces the maximum number of pairs at each processor to S. Note that the remaining (g, v)-pairs are now in sorted order.

[Phase 2] We now perform the following series of steps repeatedly until there are no more (g, v)-pairs to be considered:

1. Find the minimum g-value among the remaining (g, v)-pairs in the whole hypercube.

2. Find the result of applying F to the list of v-values of all the (g, v)-pairs with this g-value and remove these pairs.
3. Send the resulting pair to the highest-numbered processor not yet holding $\lceil S/P \rceil$ representative pairs.

Since the initial ${\beta_1}^+(R, 1, S)$ leaves the pairs sorted at each processor, each step in Phase 2 can be accomplished with a simple tree operation. Thus, the entire phase takes time $O(S \log P)$.

Since $S < R$ and $P \leq S$, $S \log P = O(R \log S)$. Thus, this subcase can be performed in time $O(R \log S)$.

∎

4.4.2 $R \leq S$ $(N \leq PS)$

Theorem 4.4.2 *If we are given N pairs distributed evenly at the P processors of a hypercube and $R \leq S$, then we can perform ${\beta_1}^+(N, P, S)$ in time*

$$O(R \log P + T_{\mathrm{SORT}}(N, P))$$

when $P \leq S \leq N$.

Proof: The algorithm for this subcase is patterned after the $Z = N$ subcase of Theorem 4.3.2.

[Phase 1] We begin by performing $\mathrm{SORT}(N, P)$.

[Phase 2] Each processor now performs ${\beta_1}^+(R, 1, S)$ in parallel. Since the pairs are already sorted, each processor requires time only linear in the number of pairs.

[Phase 3] Following the ${\beta_1}^+(R, 1, S)$ operation of Phase 2, there are no more than $P + S \leq 2S$ remaining pairs and no more than R pairs in any given processor. We can reduce the total number of pairs to S with no more than R in any given processor, as follows.

Let us call the smallest g-value contained in a pair in the processor with address p_h, α_h, and the largest, ω_h. Call the associated pairs α-pair$_h$ and ω-pair$_h$, respectively. After the first two phases are completed, α-pair$_h$ and ω-pair$_h$ are the only pairs in p_h that can belong to groups that appear in pairs outside of p_h. Hence, we shall call these two pairs *reducible*.[1] We shall call the pairs that are not reducible, *irreducible*.

Let us view the reducible pairs as a string of length no greater than $2P$. This string is composed of contiguous substrings of pairs with common g-values. We can compact each substring to a single pair in parallel in time $O(\log P)$ using the method of Lemma 3.2.2. After this compaction, there is exactly one representative pair from each group in the hypercube.

[1] It may be that there is only one reducible pair at a processor.

[Phase 4] There are now S remaining pairs and no more than R pairs left in any given processor. Ullman ([Ull]) showed that in time $O(R \log P)$. these pairs can be distributed $\lceil S/P \rceil$ per processor.

[Phase 5] We complete the operation by sorting the pairs by g-value.

The total required time for the five phases simplifies to

$$O(R \log P + T_{\text{SORT}}(N, P)).$$

∎

4.5 Time Analysis

4.5.1 Case 1: $S \leq N \leq P$

Theorem 4.5.1 *If we are given N pairs distributed evenly at the P processors of a hypercube, then we can perform $\beta_1{}^+(N, P, S)$ in time*

$$O\left(\log P + \frac{\log^2 S}{\log P/N}\right)$$

when $S \leq N \leq P$.

Proof:

In [NS82], it is shown that if we have a P-processor hypercube and N items, where $P = N^{1+1/j}$ for j in the range $1 \leq j \leq \log N$, then we can sort the items in time $O(j \log N)$.

We know from the proof of Theorem 4.2.1 that $\beta_1{}^+(N, P, S)$ can be performed in time

$$O\left(\log P + T_{\text{SORT}}(Z, \frac{Z}{R})\right)$$

where $Z = \text{Minimum}(S^3, N)$.

If $S^3 \leq N$, then

$$T_{\text{SORT}}(Z, \frac{Z}{R}) = T_{\text{SORT}}(S^3, \frac{S^3}{R}).$$

Let $P' = S^3/R$ and $N' = S^3$. P' is then equal to $(N')^{1+1/j'}$ where $j' = \log S^3/(\log P/N)$. By the [NS82] result, $\text{SORT}(S^3, S^3/R)$ $(= \text{SORT}(N', P'))$ can be performed in time

$$O(j' \log S) = O\left(\frac{\log^2 S}{\log P/N}\right).$$

If $S^3 > N$, then

$$T_{\text{SORT}}(Z, \frac{Z}{R}) = T_{\text{SORT}}(N, P).$$

By the [NS82] result, $T_{\text{SORT}}(N, P)$ is

$$O\left(\frac{\log^2 N}{\log P/N}\right).$$

This bound can also be expressed as

$$O\left(\frac{\log^2 S}{\log P/N}\right).$$

Thus, the overall time required is

$$O\left(\log P + \frac{\log^2 S}{\log P/N}\right).$$

■

4.5.2 Case 2: $S \leq P \leq N$

Theorem 4.5.2 *If we are given N pairs distributed evenly at the P processors of a hypercube, then we can perform $\beta_1{}^+(N, P, S)$ in time*

$$O(R \log S + \log P + \frac{R}{r} \log^2 S)$$

when $S \leq P \leq N$.

Proof: In Section 4.3.1, it was shown that if $R > S$, we can perform $\beta_1{}^+(N, P, S)$ in time

$$O(R \log S + \log P + T_{\text{SORT}}(S, S))$$

when $S \leq P \leq N$. Since $T_{\text{SORT}}(S, S) = O(\log^2 S)$ and $\log S < S < R$, the above expression can be simplified to

$$O(R \log S + \log P).$$

In Section 4.3.2, it was shown that if $R \leq S$ we can perform $\beta_1{}^+(N, P, S)$ in time

$$O\left(R \log S + \log P + T_{\text{SORT}}(Z, Z/R)\right)$$

where $Z = \text{Minimum}(S^2, N)$ and $S \leq P \leq N$.

[CS88] demonstrate an algorithm for sorting N items on P processors when $N = P^{1+1/c}$ for a constant c. [CS88] also demonstrate how their algorithm can be implemented on a shuffle-exchange network in time $O(c\ P^{1/c} \log P)$ and on a hypercube in time $O(c^2\ P^{1/c} \log P)$.[2] In Appendix A, we show that the hypercube sort can also be implemented in time $O(c\ P^{1/c} \log P)$, using a simple modification of the [CS88] results.[3] These sorts require that $N = P^{1+1/c}$ for a constant $c \geq 1/2$, and that $P \geq (4c^2 + c)^{2c}$.

[2] Although c is a constant, we shall note it explicitly in the big-O notation. The generalization of the [CS88] sorting results, namely, the problem of sorting N items on a P-node hypercube in time

$$O\left(\frac{\frac{N}{P}\log^2 N}{\log \frac{N}{P}}\right)$$

when $N \geq P$, is open.

[3] If $N = P^{1+1/c}$, then $c\ P^{1/c} \log P = \Theta(R/r \log^2 N)$.

Using the result of Appendix A, we can see that if $Z = S^2$,

$$T_{\text{SORT}}(Z, Z/R) = T_{\text{SORT}}(S^2, S^2/R) = O(\frac{R}{r}\log^2 S).$$

If $Z = N$, then

$$T_{\text{SORT}}(Z, Z/R) = T_{\text{SORT}}(N, N/R) = O(\frac{R}{r}\log^2 N).$$

This bound can also be expressed as

$$O(\frac{R}{r}\log^2 S).$$

Since $r \leq \log S$, the bound of Section 4.3.2 simplifies to

$$O(\frac{R}{r}\log^2 S + \log P).$$

Thus, the algorithm for the general case takes time

$$O(R\log S + \log P + \frac{R}{r}\log^2 S).$$

■

4.5.3 Case 3: $P \leq S \leq N$

Theorem 4.5.3 *If we are given N pairs distributed evenly at the P processors of a hypercube, then we can perform $\beta_1{}^+(N, P, S)$ in time*

$$O(R\log S + \frac{R}{r}\log^2 S)$$

when $P \leq S \leq N$.

Proof: In Section 4.4.1, it was shown that if $R > S$ we can perform $\beta_1{}^+(N, P, S)$ in time $O(R\log S)$ when $P \leq S \leq N$.

In Section 4.4.2, it was shown that if $R \leq S$, we can perform $\beta_1{}^+(N, P, S)$ in time

$$O(R\log P + T_{\text{SORT}}(N, P))$$

when $P \leq S \leq N$. From Appendix A, we know that

$$T_{\text{SORT}}(N, P) = O(\frac{R}{r}\log^2 N)$$

if those constraints resulting from the [CS88] algorithm, mentioned in the previous section, are satisfied. Now since $R \leq S$, it follows that $N \leq SP \leq S^2$. Thus,

$$T_{\text{SORT}}(N, P) = O(\frac{R}{r}\log^2 S).$$

Hence, the time bound for the algorithm of Section 4.4.2 is

$$O(\frac{R}{r}\log^2 S).$$

The algorithm for the general case takes time

$$O(R\log S + \frac{R}{r}\log^2 S).$$

■

4.6 Performing the Beta Operation when S is Unknown

In Section 3.4.1, we considered the problem of creating an algorithm, Algorithm B, for solving 'problem $\mathcal{P}$ not given output size S,' using as a subroutine, an algorithm, Algorithm A, for solving 'problem $\mathcal{P}$ given output size S.' In Theorem 3.4.1, we showed that if the running time of Algorithm A was $t(N, Q)$ (where Q is a given bound on the output size), then Algorithm B could be constructed with a running time that was $\Theta(t(N, Q))$. We then applied this result to the hypercube and MOT implementations of ${\beta_1}^+(N, N, S)$.

Theorem 3.4.1 can also be applied to the various implementations of ${\beta_1}^+(N, P, S)$ presented in this chapter. Let us modify conditions (i)-(iv) and (a)-(b) of Section 3.4.1 by replacing $t(N, Q)$ with $t(N, P, Q)$. If we have an algorithm that satisfies the modified conditions (i)-(iv) and an associated guess sequence that satisfies the modified conditions (a)-(b), then Theorem 3.4.1 holds, as before, with $t(N, Q)$ replaced by $t(N, P, Q)$ and $\tilde{t}(N, Q)$ replaced by $\tilde{t}(N, P, Q)$. Lemma 3.4.1 can then be modified to produce Lemma 4.6.1, below.

Lemma 4.6.1 *Let $t(N, P, Q) = c(G(N, P) + F(N, P, Q))$ where c is a constant and $F(N, P, Q)$ is a function that is monotonically increasing in Q. The guess sequence*

$$g_0 = 1,\ g_i = min\{x \mid F(N, P, x) = ((2^i - 1)G(N, P))\},\ i > 0$$

satisfies the modified conditions (a)-(b) with $c_1 = c_2 = 2$.

The algorithms of Sections 4.2, 4.3, and 4.4 satisfy the modified conditions (i)-(iv) and the associated running times are of the form described in Lemma 4.6.1. It follows, therefore, from the modified Theorem 3.4.1 and Lemma 4.6.1, that the running times of these algorithms do not increase by more than a constant factor if S is not known beforehand.

Chapter 5

The Multi-Prefix Operation

5.1 Overview

In this chapter, we shall consider the efficient implementation of another primitive parallel operation of the Connection Machine, the multi-prefix operation. In Section 5.2, we first define the multi-prefix operation. In Section 5.3, we demonstrate how β_2 can be implemented efficiently by preserving a record of the paths of the data items during the implementation of the simpler formulation of the beta operation, β_1.[1] In Section 5.4, we then show that by saving additional information about the partial computations calculated during our implementation of β_2, we can implement the multi-prefix operation efficiently on a hypercube. We shall show that our implementation of the multi-prefix operation can be composed from the same set of primitives that was used in our implementation of β_2, without increasing the use of any of the primitives by more than a constant factor. One consequence of this result is that the two implementations will require the same amount of time. We can use these two implementations as paradigms for implementing these two primitives on other networks. We first define the multi-prefix operation.

5.2 Definition

The multi-prefix operation is similar in form to the β_2 operation. Let p_{ij} be the jth processor in $proc\left(B_i\right)$. Let l_{ij} be the ordered list of v-values from the first j pairs in B_i. If the input function is F, the multi-prefix operation computes $y_{ij} = F/l_{ij}$, for $i \in G$, $1 \leq j \leq |B_i|$ and leaves (i, y_{ij}) in p_{ij}.

Example 5.2.1 *Let F be '+' and N be 5. Let the (g,v)-pairs in the input be:*

$$proc\ 1\text{: } (9,6) \quad proc\ 2\text{: } (2,4) \quad proc\ 3\text{: } (3,4) \quad proc\ 4\text{: } (9,5) \quad proc\ 5\text{: } (2,5)$$

Performing the multi-prefix operation on these pairs produces the output:

$$proc\ 1\text{: } (9,6) \quad proc\ 2\text{: } (2,4) \quad proc\ 3\text{: } (3,4) \quad proc\ 4\text{: } (9,11) \quad proc\ 5\text{: } (2,9)$$

[1] For convenience, we shall refer to $\beta_1(S)$ and $\beta_2(S)$ as β_1 and β_2 in this chapter.

5.3 Implementation of β_2

5.3.1 Overview

We start by reviewing the Section 3.2 implementation of the β_1 operation. In the naive approach to implementing β_1, we would start by sorting all the pairs by their group values. We could then easily reduce each group to a single representative value and route these values to their final destinations. The problem with this approach is that the initial sort consumes too much time.

The Section 3.2 implementation of β_1 takes advantage of the fact that the number of pairs decreases as a result of the combination that occurs as the output is calculated. This decrease in the amount of data allows the combination process to speed up as the algorithm progresses.[2] During each step of the implementation, the data are separated into subcubes of size S^4, and β_1 is performed on the pairs in each subcube to produce S new pairs. These new pairs are themselves grouped together in subcubes for the next step. We can view the implementation as creating a circuit, each gate of which is an S^4-input, S-output gate that performs β_1. The S pairs that are output by the final gate in this circuit represent the result of performing β_1 on the N-node hypercube.

In this section, we shall show how β_2 can be implemented with the same efficiency as β_1. The implementation of β_2 that we present combines pairs in the same way, and creates the same circuit as that used in the implementation of β_1. In this case, however, the circuit will be saved so that the pairs produced by the final gate can be routed back to their initial sources. Our implementation of β_2 will be composed of the same set of primitives that was used in our implementation of β_1.

We shall use the same notation as in Section 3.2. It is assumed below that S is given. For convenience, let us call the implementation of β_1 presented in Section 3.2, *Implementation-β_1*, and the implementations of β_2 and the multi-prefix operation presented below, *Implementation-β_2* and *Implementation-MP*, respectively. Recall that θ is the binary string of s 1's (i.e., $S-1$), and ϕ, the binary string of $s-1$ 1's, followed by a zero (i.e., $S-2$).

5.3.2 The General Step

The general step of both *Implementation-β_1* and *Implementation-β_2* is the same, except that at the end of the sort and compact stages, we now store pairs for future use. More specifically, at the end of these two stages, each processor stores the source address of the pair it is holding, in a processor whose address is Hamming distance one away. We shall show later that our choice of storage locations ensures that the circuit can be stored with no node needing more than a constant amount of storage.

As in *Implementation-β_1*, in every step of *Implementation-β_2*, we conceptually break the hypercube into a number of smaller subcubes and perform several primitive operations in each subcube in parallel. Each subcube corresponds to a gate in the circuit being stored and each step corresponds to performing the gate operations on a single level of the circuit. In the steps of Phase 3, the addresses in the stored circuit are used to route the output values back to the initial sources.

[2] This approach is similar to the filtration paradigm of [HMS84].

5.3.3 *Implementation-β_2* - Phase 1

In this phase, we create the first level of the circuit. We conceptually break the hypercube into a number of subcubes of size S^4. Each subcube corresponds to a first level gate. Each gate has S^4 input pairs and S output pairs. Thus, in this phase, the number of pairs is reduced from N to N/S^3. The output pairs from this phase reside in N/S^4 subcubes of size S.

Instead of keeping just a (g, v)-pair at the processors, each processor will maintain a (g, v, p)-triple, where p is a processor number. The g- and v- slots are filled with the values from the (g, v)-pair. In every step, each processor puts its address in the p-slot.

By convention, k is 0 for the one step of this phase; thus, the hypercube addresses are of the form $u_0v_0w_0x_0$, where $|u_0| = D - 4$, $|v_0| = 3$, $|w_0| = 0$, and $|x_0| = 1$. The nodes in a given subcube have addresses of the form $U_0\underline{V_0}\,\underline{X_0}$. There are, therefore, $|u_0|$ first level gates in the corresponding circuit. The U_0-th such gate has inputs of the form $U_0\underline{V_0}\,\underline{X_0}$ and outputs of the form $U_0\theta^3\underline{X_0}$.

For each subcube, H, we perform the following three stages:

Sort Stage. Using odd-even merge sort we sort the triples by g-value. Each processor then stores its triple in its local memory.

Reduce Stage. We next invoke Lemma 3.2.2 to reduce each $B_i(H)$ to a single triple.

Compact Stage. As in *Implementation-β_1*, we route the final pairs in each S^4-node subcube to the S-node subcube with addresses of the form $U_0\theta^3\underline{X_0}$. All processors not holding a final triple set their triples to the null value. The final triples represent the output of the first level gates.

We now save the information that, during Phase 3, allows us to route the output values back to the initial sources. In what follows, let us call a processor holding a non-null triple, a *live* processor. Let us call the S processors with the highest addresses in H, the *live set* of H. Every live processor with an address of the form $U_0\theta^3X_0$ now stores a copy of its triple in the local memory of the processor that is Hamming distance one away, with address $U_0\theta^2\phi X_0$. Let us use the shorthand notation, ls_{comp}, to refer to a live set produced by the compact stage. As noted above, the ls_{comp}'s have addresses of the form $U_0\theta^3\underline{X_0}$.

5.3.4 *Implementation-β_2* - Phase 2

In this phase, we complete the building of the circuit. The number of triples is reduced to S by repeated applications of the general step. There are $D - 4$ steps. Step k $(1 \leq k \leq D - 4)$ is performed as shown below.

As in Phase 1, we shall view the processor addresses as being of the form $u_kv_kw_kx_k$ during Step k, where $|u_k| = D-4-k$, $|v_k| = 3$, $|w_k| = k$, and $|x_k| = 1$. At the start of Step k, the processors with addresses of the form $U_{k-1}\theta^{k+2}X_{k-1}$ are live. We start Step k by breaking the N/S^k processors whose addresses are of the form $U_kV_k\theta^kX_k$ into N/S^{k+4} subcubes of S^4 nodes each of the form $U_k\underline{V_k}\theta^k\underline{X_k}$. Note that with this choice of partitioning the hypercube, each subcube contains at most S^2 live processors. Thus, each corresponding gate has S^2 or fewer non-empty inputs and S outputs.

As in Phase 1, there are $|u_k|$ gates on level k in the corresponding circuit. The U_k-th such gate has inputs of the form $U_k\underline{V_k}\theta^k\underline{X_k}$ and outputs of the form $U_k\theta^{k+3}\underline{X_k}$.

Next, each live processor in one of these subcubes creates a new triple by copying the g- and v-values from the triple it received in the previous compact stage into the new g- and v-slots, and putting its address in the p-slot. For each S^4-node subcube, H, of the form described above, we perform the following three stages:

Sort Stage. We can use the Nassimi-Sahni sort [NS82] to sort the at most S^2 (g, v, p)-triples in each subcube by g-value. When the sorting is complete, each live processor with address of the form $U_{k-1}\theta^{k+2}X_{k-1}$ stores its triple in the processor with address $U_{k-1}\theta\phi\theta^kX_{k-1}$.

Reduce Stage. We reduce each $B_i(H)$ to a single triple, using the reduce stage of *Implementation-*β_1.

Compact Stage. We then compact the final triples into the processors with the highest-numbered addresses in H. When the compaction has been completed, each live processor with address of the form $U_k\theta^{k+3}X_k$ stores a copy of its triple in the processor with address $U_k\theta^2\phi\theta^kX_k$. As in Phase 1, the copy will be used to enable us to route the output values back to the initial sources during Phase 3. Following the notation of Phase 1, we shall refer to the live sets produced by the compact stage as ls_{comp}'s. As noted above, the ls_{comp}'s have addresses of the form $U_k\theta^{k+3}\underline{X_k}$.

5.3.5 *Implementation-*β_2 - Phase 3

Each step of Phases 1 and 2 corresponds to performing the gate operations on a single level of the circuit. To enable us to "reverse" the gate operations on a level, we stored triples twice during each step, once during the sort stage, and once during the compaction stage. The p-values of the triples stored in the sort stage allow us to undo the sort. The p-values of the triples stored in the compaction stage allow us to reverse the compaction and undo the effects of the associated reduction stage.

It is important to keep in mind that all the live sets retrieved during this phase will be retrieved in the exact reverse order in which they were stored; e.g., we start Step k of Phase 3 by retrieving the ls_{comp}'s that were stored in the compaction stage of Step k of Phase 2. The retrieved live sets allow us to route the output values back down the circuit, one level at a time. As in the previous phases, we shall view the processor addresses as being composed of four fields, $u_kv_kw_kx_k$, during Step k. There are $D-3$ steps. We shall count down from $D-4$ to 0. Step k ($D-4 \geq k \geq 0$) is performed as shown below.

Un-compact Stage. We start Step k by retrieving the p-values of the ls_{comp}'s that were stored in the compaction stage of Step k of Phase 2. The processor with address $U_k\theta^{k+3}X_k$ replaces the p-value of its triple with the p-value from the triple of the processor with address $U_k\theta^2\phi\theta^kX_k$. The new p-value in each triple is the address of this triple prior to the compact stage of the Step k of Phase 2. With a monotone route, we can send each triple to its pre-compaction location.

Un-reduce Stage. Every processor that received a triple during the monotone route of the un-compact stage now calls itself a leader. Using two distribution-from-leaders operations, each leader can now distribute its (g, v, p)-triple to the processors in its $proc\Big(B_i(H)\Big)$ on either side.

Un-sort Stage. Finally, we retrieve the live sets stored in the sort stage of Step k of Phase 2. Let us call these sets the ls_{sort}'s. The live triples now replace their p-values with the p-values of the retrieved ls_{sort}'s. We can then reverse the sort of Step k of Phase 2 by sorting the triples by p-value.

The last step reverses the single step of Phase 1 and leaves (i, y_i) in each member of $proc\Big(B_i(H)\Big)$, as desired.

Note that the addresses of the live sets stored during Step k of Phase 2 have a ϕ in the $(k+2)$nd least significant digit of their addresses. The addresses of the live sets stored in Step j, for $j > k$, disagree with the addresses of the live sets stored in Step k in the $(k+2)$nd digit. The addresses of the live sets stored in Step j, for $j < k$, disagree with the addresses of the live sets stored in Step k in the $(j+2)$nd digit. It is clear, therefore, that no more than a constant number of triples are stored in any processor in the course of the implementation.

5.3.6 Time Analysis

Phases 1 and 2

Each act of storing (or retrieving) triples takes constant time, as every processor stores its triple either in local memory or in the local memory of a processor whose address is Hamming distance one away. Aside from operations taking constant time, the running times of these two phases are identical to their counterparts in *Implementation-β_1*. From Section 3.2 we know that these two phases can be performed in time

$$O(\log N + T_{\text{SORT}}(S, S)).$$

Phase 3

The sort used in the un-sort stage of Step k ($k > 0$) can be performed with the Nassimi-Sahni algorithm in time $O(\log S)$. The sort used in the un-sort stage of Step 0 can be performed in time $O(T_{\text{SORT}}(S, S))$. The remaining two stages require only a constant number of distribution-from-leaders operations and monotone routes, each of which takes time $O(\log S)$.

Step k ($k > 0$) takes time $O(\log S)$ and there are $O(\log N / \log S)$ steps. Step 0 takes time

$$O(\log S + T_{\text{SORT}}(S, S)).$$

Thus, Phase 3 can be performed in time

$$O(\log N + T_{\text{SORT}}(S, S)).$$

The overall time used by the implementation is, therefore,

$$O(\log N + T_{\text{SORT}}(S, S)).$$

5.4 Implementation of the Multi-Prefix Operation

5.4.1 Overview

Implementation-MP is created by adding a number of simple modifications onto *Implementation-β_2*. Specifically, there are four important modifications.

- We first perform a pre-processing phase, Phase 0, that allows us to assume that the g-values of the initial pairs range from 0 to $S-1$.
- In Phase 1, we calculate the partial prefixes in each $B_i(H)$ instead of simply reducing each $B_i(H)$ to a single value.
- At the end of every compact stage in *Implementation-β_2*, we routed the remaining pairs in each S^4-subcube, H, to the S highest-address processors (the live set of H). In the compact stages of *Implementation-MP*, we route the remaining pair from group i (if there is one) to the ith processor in the live set.
- In every step of Phase 1 and 2, there is an additional stage, the distribution stage. During each distribution stage, we shall store the nodes of a tree of (g,v,p)-triples. These nodes will be retrieved in Phase 3 to produce the correct values of y_{ij}.

In Section 2.1, we stated that as input to either variant of the beta operation, we are given an associative, commutative binary function F and N pairs, $(g_0,v_0),\ldots,(g_{N-1},v_{N-1})$. By introducing minor modifications to our Section 3.2 implementation of β_1 and our Section 5.3 implementation of β_2, we can create implementations that do not require F to be commutative. More specifically, if we use sorts that are stable and preserve the ordering during the simple tree operations of Step 3 of Lemma 3.2.2, the implementations presented for β_1 and β_2 will work correctly even if the input function, F, is not commutative. Similarly, we shall show that for the implementation of the multi-prefix operation presented below, F is not required to be commutative.

5.4.2 *Implementation-MP* - Phase 0

Given an instance of the multi-prefix operation, we store the input pairs and run *Implementation-β_2* to the end of Phase 2. At this point, the number of triples has been reduced to S and the circuit has been stored for use in Phase 3. With a simple prefix operation, each triple can determine its rank (from 0 to $S-1$). At the end of Phase 3, each triple replaces its g-value with its rank and restores its v-value. We may now assume that the g-values of the initial pairs range from 0 to $S-1$.

5.4.3 *Implementation-MP* - Phase 1

In this phase, we create the first level of the circuit. As in *Implementation-β_2*, we break the hypercube into a number of subcubes and reduce the number of triples in each subcube from N to N/S^3. For

each subcube, H, we also calculate the partial prefixes in each $B_i(H)$. We use the same three stages of Phase 1 in *Implementation-β_2* with the following modifications. A fourth stage is also added below: the distribution stage.

Sort Stage. In order to ensure that the correct partial prefixes are calculated in the next stage, it is necessary that the sort be stable. We can achieve a stable sort by using two keys. We sort the triples by g-value with an odd-even merge sort; however, if two triples have equal g-values, we compare p-values. Each processor then stores its triple in its local memory.

Reduce Stage. We can compute the partial prefixes for each $B_i(H)$ by invoking a trivially modified version of Lemma 3.2.2. To start, every processor stores its address in the p-slot of its triple. We then apply the first two steps of Lemma 3.2.2. Upon completion of the second step, each processor in CB_i contains, at most, four triples. Using simple tree operations, we can now compute the partial prefixes for each $B_i(H)$ in parallel, taking care to preserve the ordering of the triples. We store the resulting values in the v-slots of each triple. Using the addresses stored at the beginning of this stage, we can route the triples back to their original sources by reversing the monotone routes of Step 2 of Lemma 3.2.2. Each processor then stores its triple.

Compact Stage. If there is a pair in H from group i after the reduction stage, we route it to the ith processor in H's live set.

Distribute Stage. When the reduction stage has been completed, we shall have calculated the value of F applied to the $B_i(H)$'s in each separate subcube, H. To compute the y_{ij}'s, we must combine the value of F applied to the $B_i(H)$'s in each subcube with the values of F applied to the corresponding $B_i(H)$'s in all subcubes with lower addresses. In this stage, we shall create a tree of (g, v, p)-triples with each node in the tree representing the set of values produced by applying F to the $B_i(H)$'s in a number of H's. During Phase 3, the nodes of this tree will be recalled in the reverse order in which they were produced in order to form the y_{ij} values.

Recall that we referred to the live sets produced by the compact stage as ls_{comp}'s. At the completion of the compaction stage in *Implementation-β_2*, a copy of each ls_{comp} with address of the form $U_0\theta^3\underline{X_0}$ was stored in the processors with addresses of the form $U_0\theta^2\phi\underline{X_0}$. At the beginning of the distribution stage, each processor makes a copy of its stored ls_{comp} triple. We shall call these new copies, ls_{dist}'s, by analogy.

The distribution stage is basically a prefix operation where the values operated on are S-tuples. More specifically, when we complete this stage, each ls_{dist} will contain the S or fewer values representing the results of F applied to the triples from all ls_{dist}'s in the containing block whose addresses are strictly lower. Note that this partial prefix operation is not inclusive; i.e., at the end of the stage, the new value of ls_{dist} does not include its old value.

We start by grouping the subcubes together in *blocks* of S subcubes each. Consider a particular block, L. To start, every processor in L that holds a triple stores its address in the p-slot. We next send every $B_i(H)$ to a different S-node hypercube in L as follows. If there is a triple from group i in the jth ls_{dist}, it is sent to the jth position in the ith S-node hypercube in L.

Note that the triples in such a hypercube are ordered by their original source addresses. Those processors that did not receive a triple as a result of the last operation now create a dummy triple with v-value 0. Using simple tree operations, we can now compute the non-inclusive partial prefixes in each S-node subcube. The resulting v-values are stored in the v-slots of the associated triples. Finally, we route all triples (including the dummy triples) in a given S-node subcube back to the S ls_{dist}'s. The triple in the jth position in the ith such subcube is sent to the ith position in the jth ls_{dist}. The triples computed in this stage are stored in local memory.

5.4.4 *Implementation-MP* - Phase 2

In this phase, the number of triples is reduced to S^2 by repeated application of the general step.[3] At the end of the phase, all the information for determining the y_{ij}'s is stored in the tree of triples. There are $D-5$ steps. The first three stages of Step k ($1 \leq k \leq D-5$) are the same as those of Step k of Phase 2 of *Implementation-β_2*, with two modifications. First, in the monotone route of the compaction stage, we route the remaining pair from group i (if there is such a pair) to the ith processor in H's live set. Secondly, as in Phase 1, after the compact stage, there is an additional stage, the distribution stage. The distribution stage of Step k has the same form as in Phase 1. As before, the sort of the sort stage must be stable; we achieve a stable sort by using the g-value as a primary key and the p-value as a secondary key. The ls_{dist}'s have addresses of the form $U_k\theta^2\phi\theta^k\underline{X_k}$.

5.4.5 *Implementation-MP* - Phase 3

During Phase 2, we stored triples three separate times during each step. As in *Implementation-β_2*, the p-values of the triples stored during the sort and compact stages were used to create a circuit that allows us to route the output results back to the appropriate processors. In addition, we also created a tree of triples representing values of F applied to various subcubes.

More concretely, the end value of a particular y_{ij} is determined as follows. At the beginning of the distribution stage of Step k of Phase 2, the ls_{dist} with address $U_k\theta^2\phi\theta^k\underline{X_k}$ represents F applied to the v-values from the subcube of N/S^{D-4-k} processors with addresses of the form $U_k\underline{V_k}\,\underline{W_k}\,\underline{X_k}$. At the end of this stage, each ls_{dist} represents all the processors that were represented by those ls_{dist}'s in the containing block with lower addresses. The value of y_{ij} is calculated by combining those ls_{dist}'s that represent the processors with addresses lower than p_{ij}. As in *Implementation-β_2*, all the live sets retrieved during this phase are retrieved in the reverse order that they were stored. Note that the retrieval process maintains the correct ordering among the triples. There are $D-4$ steps. We count down from $D-5$ to 0.

We start Step k ($D-5 \geq k \geq 1$) by retrieving the ls_{dist}'s that were stored in the distribution stage of Step k of Phase 2. The ls_{dist} with address of the form $U_k\theta^2\phi\theta^k\underline{X_k}$ is retrieved into the ls_{comp} with address of the form $U_k\theta^{k+3}\underline{X_k}$. The v-value from each ls_{dist} triple is transferred to the

[3] For technical reasons, there is one fewer step in this Phase 2 than in the Phase 2 of *Implementation-β_2*.

v-slot of the corresponding ls_{comp} triple. Afterwards, the reverse routing of the triples of the ls_{comp}'s on level k of the circuit is identical in form to Step k of Phase 3 of *Implementation-β_2*.

Finally, we repeat these four stages one more time with k equal to 0, in order to reverse Phase 1. One modification is introduced in the last step. At the end of the un-reduce stage, the triples stored during the reduce stage of Phase 1 are retrieved and the v-value of each retrieved triple is combined with the v-value of the triple already contained in the processor. The following un-sort stage leaves y_{ij} in p_{ij} as desired, and completes the implementation.

5.4.6 Time Analysis

In Section 5.3, we presented an efficient implementation of β_2 in terms of a number of simple primitives. Aside from $O(1)$ operations, four modifications are applied to *Implementation-β_2* to create *Implementation-MP*. These modifications are enumerated in Section 5.4.1. We now show that these modifications do not increase the use of any of the simple primitives by more than a constant factor.

The first modification is the addition of a pre-processing phase, Phase 0. In this phase, we ensure that the g-values of the initial pairs range from 0 to S-1. This phase requires us to perform the three phases of *Implementation-β_2* plus a single tree operation on S items. The second modification replaces one tree operation with another in the reduction stage of Phase 1 of *Implementation-β_2*. The third modification simply changes the destination addresses in the monotone route of the compact stage. The fourth modification, the addition of the distribution stage, uses only a constant number of the primitives already required by the sort and reduce stages.

5.5 Conclusion

In developing *Implementation-MP*, we first reviewed the implementation of β_1 from Section 3.2. A circuit was created in the course of this implementation. In Section 5.3, we showed that this circuit could be used to create an implementation for β_2. We then showed in Section 5.4 that the implementation of β_2 could be modified to create an implementation of the multi-prefix operation without increasing the complexity of the computation. One consequence of these results is that the implementations of the β_2 and the multi-prefix operations presented in this chapter can be used as paradigms for implementing these two primitives on other networks. Thus, for example, we can modify the Section 3.3 MOT implementation of β_1 to obtain MOT implementations of β_2 and the multi-prefix operation with the same running times.

In *Implementation-β_2* and *Implementation-MP*, we assumed that S was known. Nonetheless, from the result of Section 3.4, it follows that the same time bounds hold even if S is not known beforehand.

The multi-prefix operation can be generalized by making F dependent on the g-value. If there is a different function, F_g, associated with each group, the above analysis remains valid.

Chapter 6

Applications

6.1 Simulating Beta Operations (with Other Beta Operations)

6.1.1 Overview

In Chapter 1, we stated that a primitive operation should balance efficiency and generality. In this section, we demonstrate this balance for the beta operation by simulating ${\beta_1}^+(N, N, S)$ using only applications of $\beta(S')$ (for $S' < S$). The generality is evidenced by the fact that we need no other primitives. The efficiency is evidenced by the fact that the time requirements match those of the the most efficient known algorithm for performing ${\beta_1}^+(N, N, S)$. We shall show that if $S' = S^{1/c}$ for some constant $c > 1$, then the above simulation can be performed on an N-node hypercube with $O(c^2 + \log N / \log S')$ applications of $\beta(S')$.[1] For a constant $c > 1$, the Section 3.2 algorithm takes as long in the case of $S = N^{1/c}$ output groups as it does in the case of $S = N$ output groups. The simulation of this section shows that the two cases are "just as hard".

In Section 6.1.2, a number of auxiliary theorems needed for our result are established. The algorithm for the simulation is given in Section 6.1.3. In Section 6.1.4, the total required time is calculated.

6.1.2 Auxiliary Theorems

Below, we present five simple auxiliary theorems needed for establishing the main result of this section (Theorem 6.1.6). Let $\log S'$ be called s'. For convenience, we shall call the binary string composed of s' 1's (i.e., $S' - 1$), μ, and the binary string composed of $s' - 1$ 1's (i.e., $S'/2 - 1$), γ. As before, the binary string composed of s 1's (i.e., $S - 1$) will be called θ. We shall assume below that there is never more than one pair per processor.

The first theorem demonstrates that sorting can be performed using beta operations.

[1] As in Sections 4.5.2 and 4.5.3, we shall note c explicitly in the big-O notation although it is a constant.

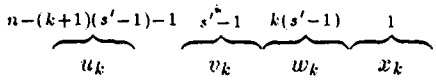

Figure 6.1: Composition of Hypercube Addresses.

Theorem 6.1.1 *N items can be sorted stably on an N-processor hypercube with one application of $\beta_1{}^+(N,N,N)$.*

Proof: Let processor p_j hold item κ_j initially. Each processor forms a (g,v)-pair with the g-value set to (p_j,κ_j) and the v-value set to 0. A given g-value, (p_j,κ_j), is less than another g-value, $(p_{j'},\kappa_{j'})$, if either $\kappa_j < \kappa_{j'}$ or $\kappa_j = \kappa_{j'}$ and $p_j < p_{j'}$. Clearly no two g-values are equal. Thus, the N items can be sorted stably with one application of $\beta_1{}^+(N,N,N)$. ∎

If the input (g,v)-pairs were initially sorted, we could quickly reduce the number of pairs with repeated applications of $\beta_1(S')$. This concept is expressed as Theorem 6.1.2.

Theorem 6.1.2 *Suppose we are given an N-node hypercube holding N (g,v)-pairs that are sorted by g-value. Then every B_i can be reduced to a single pair using $O(\log N/\log S')$ applications of $\beta_1(S')$.*

Proof: Let $\log N$ be n and η be $\lceil \frac{n-1}{s'-1} \rceil$. The reduction of the B_i's is achieved with η steps. We shall view the processor addresses as being composed of four fields, $u_k v_k w_k x_k$, where k is the number of the step we are performing. The lengths of fields u_k and w_k are functions of k. The addresses are composed as shown in Figure 6.1.

In each step of the algorithm presented below, we shall conceptually break the N-node hypercube into a number of smaller subcubes of size S'. We shall reduce the number of pairs by performing $\beta_1{}^+(S',S',S')$ in each subcube in parallel. We shall distinguish between two different sets of pairs that remain after the reduction. Recall that the pairs are sorted by g-value. Consider a subcube, H. After the reduction, only the first and last pairs contained in H can belong to groups that appear in pairs outside of H. As in Section 4.4.2, we shall call these two pairs *reducible*, since they can be combined with other pairs in later steps.[2] We shall call the $S'-2$ or fewer pairs that appear between the reducible pairs *irreducible*.

At the beginning of Step k, the remaining reducible pairs will be broken into subcubes of size S' with addresses of the form $U_k \underline{V_k}\, \gamma^k \underline{X_k}$. Step k $(0 \le k < \eta - 1)$ is performed as shown below.

Reduce. We perform $\beta_1{}^+(S',S',S')$ on each S'-node subcube with addresses of the form $U_k \underline{V_k}\, \gamma^k \underline{X_k}$.

Compaction. We want to route the two reducible pairs to the two highest addresses in H and the $S'-2$ or fewer irreducible pairs to the lowest addresses in H. This routing can be performed simply with four $\beta_1(S')$ operations. The highest-address processor stores the reducible pair it holds. We then perform $\beta_1{}^-(S')$ to move the pairs to the lowest-numbered processors in H.

[2] There will be only one reducible pair if all S' pairs in H come from the same group.

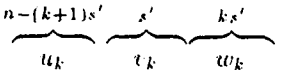

Figure 6.2: Composition of Hypercube Addresses.

The lowest-address processor, while remembering its true g-value, sets its g-value to infinity. Another $\beta_1^-(S')$ then moves the irreducible pairs to the lowest addresses in H while moving the lower-numbered reducible pair to the first processor after the irreducible pairs. Two more $\beta_1(S')$'s suffice to route the lower-numbered reducible pair to the second highest location in H (first move the pair to the highest location). The reducible pairs are now contained in locations $U_k\gamma^{k+1}\underline{X_k}$. Note that the reducible pairs maintain their sorted order.

After $\eta - 1$ steps, there are S' or fewer remaining reducible pairs in the S'-node subcube with addresses of the form $\underline{V_{\eta-1}}\,1^{n-s'}\underline{X_{\eta-1}}$. We perform one more $\beta_1^+(S', S', S')$ on this subcube. All the pairs are now irreducible. Since no two irreducible pairs can have the same g-value, every B_i must have been reduced to a single representative pair.

∎

We next present two theorems concerning simple tree operations on the (g, v)-pairs of a hypercube.

Theorem 6.1.3 *Suppose we are given an N-node hypercube holding N or fewer (g, v)-pairs. Then we can perform $\beta_1^+(N, N, 1)$ with $O(\log N/\log S')$ $\beta_1^+(S', S', 1)$ operations.*

Proof: The processor addresses will be viewed as being composed of three fields, $u_k v_k w_k$, where k is the number of the step we are performing. The addresses are composed as shown in Figure 6.2.

Let η be $\lceil n/s' \rceil$. There are η steps. At the beginning of Step k $(0 \le k < \eta - 1)$, the remaining pairs will be broken into subcubes of size S' with addresses of the form $U_k\underline{V_k}\mu^k$. $\beta_1^+(S', S', 1)$ is performed on each subcube. After $\eta - 1$ steps, there are S' or fewer remaining pairs in the S'-node subcube with addresses of the form $\underline{V_{\eta-1}}\,1^{n-s'}$. We perform one more $\beta_1^+(S', S', 1)$ on this subcube to reduce the remaining pairs to a single value. ∎

Theorem 6.1.4 *We can perform broadcast in an N-node hypercube using only $O(\log N/\log S')$ $\beta_2(S', S', 1)$ operations.*

Proof: The proof mirrors the form of the proof of Theorem 6.1.3. ∎

The previous two theorems are used in Theorem 6.1.5 below.

Theorem 6.1.5 *Suppose we are given an N^2-node hypercube holding N items in an N node subcube. Then we can perform* SORT(N, N^2) *using only $O(\log N/\log S')$ $\beta_1(S', S', 1)$ and $\beta_2(S', S', 1)$ operations.*

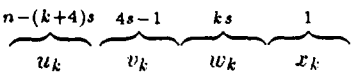

Figure 6.3: Composition of Hypercube Addresses.

Proof: We shall view the N^2-processor hypercube as an $N \times N$ matrix, p_{ij}, with the processors arrayed in order of decreasing processor number (in row-major form). Without loss of generality, we shall assume that initially, only the first row of processors contains items. Let the processor in column j, p_{1j}, hold item κ_j. We shall act as if all the items are different by using the method of Theorem 6.1.1.

We start by invoking Theorem 6.1.4 twice. We broadcast the contents of each first row processor, p_{1j}, to the column j. Then every processor, p_{jj}, broadcasts its contents to row j. Each processor, p_{ij}, compares the items received in the two broadcasts and puts a 1 in local memory if the value from p_{1j} is greater than that from p_{jj}. The sum of the comparison bits in a column is then the rank of the item from p_{1j}. The column sums can be computed by invoking Theorem 6.1.3 to perform a single ${\beta_1}^+(N,N,1)$ operation. If r_j is the rank of κ_j, then we can route κ_j to p_{1r_j} with three more broadcast operations: p_{1j} broadcasts to column j; p_{jj} broadcasts to row j; p_{jr_j} broadcasts to column r_j.

∎

6.1.3 Simulation of the Beta Operation

The auxiliary theorems of Section 6.1.2 are used in the following theorem.

Theorem 6.1.6 *Suppose we are given an N-node hypercube holding N pairs representing S different groups. Let $S' = S^{1/c}$ for some constant $c > 1$. Then the application of ${\beta_1}^+(N,N,S)$ can be simulated with $O(c^2 + \log N/\log S')$ applications of $\beta_1(S')$ and $\beta_2(S')$.*

Proof: The algorithm below is closely patterned after the Section 3.2 algorithm for computing ${\beta_1}^+(N,N,S)$.

Let $\lceil n/s \rceil - 3$ be called η.[3] There are η steps. We shall view the processor addresses as being composed of four fields, $u_k v_k w_k x_k$, where k is the number of the step we are performing. The lengths of fields u_k and w_k are functions of k. The addresses are composed as shown in Figure 6.3.

In Step k of this algorithm, we shall conceptually break the N-node hypercube into a number of smaller subcubes of size S^4 with addresses of the form $U_k \underline{V_k}\, \theta^k\, \underline{X_k}$. The algorithm is divided into two phases.

Phase 1. There is only one step in Phase 1. By convention, it will be called Step 0. In Step 0, the hypercube is broken into N/S^4 subcubes of size S^4 with addresses of the form $U_0 \underline{V_0}\, \underline{X_0}$. The pairs in each subcube can be sorted with $O(c^2)$ sorts of size $O(S')$ by invoking Corollary A.3.2

[3] If $S^2 > N$, we can perform the simulation by applying Phase 1 below to the entire hypercube.

with N equal to S and ϵ equal to 4. These sorts can be performed with $O(c^2)$ applications of $\beta_1(S')$ by invoking Theorem 6.1.1. We then invoke Theorem 6.1.2 with N equal to S^4 to reduce the number of pairs in each S^4-node subcube to at most S, using $O(s/s')$ applications of $\beta_1(S')$.[4] With one additional application of Corollary A.3.2 and Theorem 6.1.1, the remaining pairs can be routed to the S highest-address processors in their subcube with a sort.

Phase 2. Phase 2 is composed of $\eta-1$ steps. In Step k $(0 \leq k < \eta-1)$, the hypercube is broken into subcubes of size S^4 with addresses of the form $U_k \underline{V_k} 0^k \underline{X_k}$. Note that each subcube contains S^2 or fewer pairs. We can thus invoke Theorem 6.1.5 with N equal to S^2 to sort the pairs with $O(s/s')$ $\beta_1(S', S', 1)$ and $\beta_2(S', S', 1)$ operations. By Theorem 6.1.2, the number of pairs in each S^4-node subcube can then be reduced to S with $O(s/s')$ applications of $\beta_1(S')$. With one additional application of Theorem 6.1.5, the remaining pairs can be routed to the S highest-address processors in their subcube. In Step $\eta-1$, the above actions are performed on the subcube of size S^4 with address of the form $\underline{V_{\eta-1}}\, 1^{n-4s} \underline{X_{\eta-1}}$. Since $\eta-1$ is $O(n/s)$ and each step requires $O(s/s')$ applications of $\beta_1(S')$ and $\beta_2(S')$, $O(n/s')$ applications of $\beta_1(S')$ and $\beta_2(S')$ are used in this phase.

∎

As long as the sorting subroutines used are stable, the above simulation also works correctly for the version of $\beta_1{}^+(N, N, S)$ in which F is not required to be commutative.

6.1.4 Time Analysis

We have shown in Section 6.1.3 that $\beta_1{}^+(N, N, S)$ can be simulated using $O(c^2 + n/s')$ applications of $\beta_1(S')$ and $\beta_2(S')$. In this section, we show that this simulation takes the same time as the most efficient known implementation of the Section 3.2 algorithm for $\beta_1{}^+(N, N, S)$.

Theorem 6.1.7 *The application of $\beta_1{}^+(N, N, S)$ on an N-node hypercube holding N items can be simulated in time $O(\log N + \log^2 S)$ using only $\beta_1(S')$ and $\beta_2(S')$ operations.*

Proof: Consider the algorithm of Theorem 6.1.6.

Phase 1. There is only one step in Phase 1 of Theorem 6.1.6. This step begins and ends by invoking both Corollary A.3.2 and Theorem 6.1.1 to sort the pairs in each S^4-node subcube with $O(c^2)$ applications of $\beta_1(S', S', S')$. The invocation of Theorem 6.1.2 with N equal to S^4 that occurs between these sorts uses only $O(c)$ applications of $\beta_1(S', S', S')$. From Section 3.2, we know that each application of $\beta_1(S', S', S')$ takes time $T_{\text{SORT}}(S', S')$. Thus, the total time required for this phase is $O(c^2 \cdot T_{\text{SORT}}(S', S'))$. We know that there is an upper bound (possibly loose) for sorting on the hypercube model that we have been using, namely:

$$T_{\text{SORT}}(N, N) = O(\log^2 N).$$

[4] Note that $s/s' = O(c)$.

Thus,

$$c^2 \cdot T_{\text{SORT}}(S', S') = O(c^2 \cdot 1/c^2 \cdot \log^2 S).$$

The overall time for this phase of the simulation is thus $O(\log^2 S)$.

Phase 2. Phase 2 is composed of $\eta - 1$ steps. Step k begins and ends by invoking Theorem 6.1.5 to sort the pairs in each S^4-node subcube with $O(s/s')$ $\beta_1(S', S', 1)$ and $\beta_2(S', S', 1)$ operations. Both $\beta_1(S', S', 1)$ and $\beta_2(S', S', 1)$ take time $O(s')$.

By Theorem 6.1.2, the number of pairs in each S^4-node subcube can be reduced to S with $O(s/s')$ applications of $\beta_1(S')$. $\beta_1(S')$ is performed in the subcubes with addresses $U_k \underline{V_k} \theta^k \underline{X_k}$. By Theorem 4.5.1, $\beta_1(S')$ can be performed in time $O(s')$ in each cube, since the number of processors is the square of the number of items.

Since $\eta - 1$ is $O(n/s)$ and each step requires $O(s/s')$ operations that each require time $O(s')$, the time required for this phase is $O(n)$.

The total time required by the algorithm of Theorem 6.1.6 is

$$O(\log N + \log^2 S)$$

∎

6.2 Exploiting Locality

6.2.1 Overview

In this section, we present a simple application of the generalized beta operations presented in Chapter 4. In particular, using Theorem 4.5.1, we can show that if the (g, v)-pairs of the input are "bunched together," the time required for the hypercube implementation of ${\beta_1}^+(N, N, S)$ can be reduced to $\log N$.

6.2.2 A More Efficient Implementation of β_1

We demonstrate the following result:

Theorem 6.2.1 *Suppose we are given an instance of the beta operation to perform on the hypercube,* ${\beta_1}^+(N, N, S)$. *If, for a constant $c > 0$, all subcubes of size x^{1+c} contain pairs from no more than x different groups, then we can perform the beta operation in time $O(\log N)$.*

Proof: We can achieve this result by adding a simple modification to the Section 3.2 algorithm for implementing ${\beta_1}^+(N, N, S)$ on a hypercube. The modified algorithm has three phases.

Phase 1. We start by breaking the hypercube into $N/2^{\sqrt{\log S}}$ subcubes of size $2^{\sqrt{\log S}}$ and performing the ${\beta_1}^+$ operation on each of these subcubes. From Section 3.2, we know that this phase can be performed in time $O(\log S)$.

Phase 2. We next group the subcubes of Phase 1 together to form larger subcubes of size S^4. We know that after Phase 1, each cube of size S^4 has at most $S^{4/1+\epsilon}$ remaining pairs. We can now invoke Theorem 4.5.1 to perform the $\beta_1{}^+(S^{4/1+\epsilon}, S^4, S)$ operation on each subcube in time $O(\log S)$.

Phase 3. At the end of Phase 2, the condition of the remaining pairs is identical to that which pertained at the end of Phase 1 in the Section 3.2 algorithm. To complete the $\beta_1{}^+(N, N, S)$ operation, we simply apply Phase 2 of the Section 3.2 algorithm. This last phase takes time $O(\log N)$.

∎

Chapter 7

Summary

In the preceding chapters, we have examined the beta operation proposed by Hillis ([Hil85]). In Chapter 2, we presented two variant formulations of the beta operation, β_1 and β_2.

In Chapter 3, we studied the problem of implementing the beta operation efficiently on several multiprocessor networks, namely, the hypercube, mesh, and mesh-of-trees graphs. The hypercube implementation was presented first and served as a paradigm for the other implementations. We showed that all three implementations could be performed in time

$$O(\log N + T_{\text{SORT}}(S, S)),$$

where $T_{\text{SORT}}(N, P)$ is the minimum time required to sort N items on a given P processor network. The identity of the network in question is understood by context.

We next proved that any when- and where-determinate circuit that can perform beta operations for any S requires

$$AT^2 = \Omega(NS \log N).$$

This lower bound is close to the upper bound

$$AT^2 = O(N \log^2 N(S + \log^2 N))$$

realized in the mesh-of-trees implementation.

In Chapter 4, we considered the problem of implementing the beta operation efficiently when the relationship among N, P, and S was allowed to be general. We denoted the three possible relationships as:

Case 1: $S \leq N \leq P$

Case 2: $S \leq P \leq N$

Case 3: $P \leq S \leq N$

We presented implementations of the beta operation for these three cases that satisfied the following upper bounds.

Case 1:

$$O\left(\log P + \frac{\log^2 S}{\log P/N}\right)$$

Case 2:

$$O\left(R \log S + \log P + \frac{R}{\log R} \log^2 S\right)$$

Case 3:

$$O\left(R \log S + \frac{R}{\log R} \log^2 S\right)$$

In Chapter 5, we re-examined the hypercube implementation of β_1 from Chapter 3 and studied the hypercube implementation of the other variant of the beta operation discussed in Chapter 2, β_2. We showed that by creating a circuit that stores information about the partial computations that arise during the hypercube implementation of β_1, we could implement β_2 with no increase in complexity. We next presented a new primitive that generalizes the beta operation, the multi-prefix operation. We then demonstrated how our implementation of β_2 could be augmented to provide an implementation of the multi-prefix operation, again with no increase in complexity. The implementations of the β_2 and multi-prefix operations serve as paradigms for implementing these two primitives on other networks.

In Chapter 6, we presented some applications of the beta operation. We demonstrated both the efficiency and the generality of the beta operation by simulating complex beta operations using only simpler beta operations. Let $S' = S^{1/c}$ for some constant $c > 1$. We showed that the application of ${\beta_1}^+(N, N, S)$ could be simulated with $O(c^2 + \log N / \log S')$ applications of $\beta_1(S')$ and $\beta_2(S')$. This simulation entailed only a constant factor increase in the required time. Lastly, we presented a sample application of one of the results established in Chapter 4. We showed that if the input pairs are "bunched together," the time required for the hypercube implementation of ${\beta_1}^+(N, N, S)$ can be reduced.

Chapter 8

Open Questions

In Chapter 3, we studied the problem of implementing the beta operation on several multiprocessor networks. Given these implementations, we may ask more broadly how the structure of a network relates to its ability to execute this primitive operation (or any other). We may ask, for example, if there is a good way of characterizing the networks that can sort quickly. If the networks are described in a group-theoretic way, is there some way of establishing broad classes of networks whose group properties ensure the ability to perform certain primitives efficiently?

In Chapter 4, we considered the efficient implementation of ${\beta_1}^+(N, P, S)$ for all possible relationships among S, N, and P. In Sections 4.2, 4.3, and 4.4, the time requirements for these implementations were all expressed in terms of the SORT(N,P) operation. In Section 4.5, we provided actual time bounds by using the hypercube implementation of the [CS88] sorting algorithm. The [CS88] result states that if $N = P^{1+1/c}$ for some constant c, then $T_{\mathrm{SORT}}(N, P)$ is $O(c\ P^{1/c} \log P)$ on a shuffle-exchange graph and $O(c^2\ P^{1/c} \log P)$ on a hypercube. In Appendix A, we show that a slight modification to the [CS88] hypercube implementation reduces the required time to $O(c\ P^{1/c} \log P)$. Recently, [Pla88] has demonstrated that on a stronger hypercube model, SORT(N, P) can be performed in time $O(k \log P)$, when $P \log P = N^{1+1/k}$. It remains to be shown whether optimal sorting can be achieved on the stronger hypercube model for all cases where $N \geq P$. Also, the question is still open whether the [Pla88] results can be carried over to the weaker model of the hypercube that was introduced in Chapter 2.

Our implementation of ${\beta_1}^+(N, N, S)$ takes time $O(\log N + \log^2 S)$, as opposed to $O(\log^2 N)$ for the straightforward implementation. This time savings is the result of our using the first phase to reduce the amount of data and free up enough processors to perform the second phase quickly. Is this paradigm of general applicability? [HMS84] show that filtration is of use in several particular cases. Do algorithms that benefit from the use of some type of filtration share a common element?

Appendix A

Sorting on a Hypercube

A.1 Overview

In this appendix, we address the problem of sorting N items on a P-processor hypercube. In Section A.2, we consider the problem of sorting when $N = P^{1+1/c}$ for some constant c. [CS88] demonstrate a network-independent algorithm, *Cubesort*, that performs this sort. They also demonstrate how this sort can be implemented in time $O(c\ P^{1/c} \log P)$ on a shuffle-exchange graph and time $O(c^2\ P^{1/c} \log P)$ on a hypercube. We shall modify the [CS88] hypercube implementation of *Cubesort* so that it also takes time $O(c\ P^{1/c} \log P)$ (Theorem A.2.3).

In Section A.3, we consider the problem of sorting on a hypercube when $N = P$. There are many sorting algorithms that can be implemented efficiently on a hypercube whose running times match the best known upper bounds (e.g., [Bat68]). For the purposes of Section 6.1, however, we need an algorithm that sorts N items on an N-processor hypercube using only sorts of size polynomially less than N.

A.2 Performing SORT(N,P) on a Hypercube

A.2.1 Notation

We begin by reviewing the [CS88] notation. [CS88] choose two variables, M and D, satisfying three conditions:

- $M^D = N$
- $M \geq (D-1)(D-2)$
- $D \geq 3$

In what follows, the addresses of the N items will be viewed as being composed of D base-M digits. We shall view these D base-M digit item addresses as being composed of four fields,

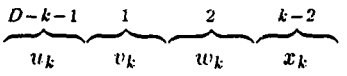

Figure A.1: Composition of Hypercube Addresses.

$u_k v_k w_k x_k$. The lengths of fields u_k and x_k are functions of k. The fields are composed as shown in Figure A.1. We shall view the $D-2$ base-M digit processor addresses as being composed of the three fields, u_k, v_k, and x_k.

[CS88] define D partitions of the items, P_k, for $1 \leq k \leq D$. Each partition, P_k, will divide the items into $\frac{N}{M^2}$ *blocks* of size M^2. In the special case where $k = 1$, P_1 divides the items into $\frac{N}{M}$ blocks. The items in one block of a particular partition, P_k, have addresses of the form $U_k V_k \underline{W_k} X_k$.[1] [CS88] employ two operations wherein all the blocks of partition P_j are sorted in parallel: *Sort_Ascending*(j) and *Sort_Mixed*(k, j). In *Sort_Ascending*(j), each block of partition P_j is sorted in ascending order. In *Sort_Mixed*(k, j), each block of partition P_j is sorted in ascending order if V_k is even and descending order if V_k is odd.

A.2.2 *Cubesort*

[CS88] present *Cubesort* (below) and prove it correct.

```
Cubesort(D)
integer D;
{
  if D = 3 then
  {
    Limit_Dirty_Cubes (D);
    Sort_Mixed (D-1,D-1);
    Merge_Dirty_Cubes (D,D);
  }
  else    /* D > 3 */
  {
    Limit_Dirty_Cubes (D);
    Limit_Dirty_Cubes (D);
    Cubesort (D-1);
    Merge_Dirty_Cubes (D,D);
  }
}
```

[1] The items of P_1 have addresses that vary only in the last base-M digit.

```
Limit_Dirty_Cubes (D)
integer D;
{
    if  D > 2  then
        Limit_Dirty_Cubes (D-1)
    Sort_Ascending (D);
}

Merge_Dirty_Cubes (D,T)
integer D,T;
{
    Sort_Mixed (D,T);
    if  T > 2  then
        Merge_Dirty_Cubes (D,T-2)
}
```

Consider the two subroutines of *Cubesort*, Limit_Dirty_Cubes (S) and Merge_Dirty_Cubes (S). We can unroll Limit_Dirty_Cubes (S) to get

```
Limit_Dirty_Cubes (D)
integer D;
{
    For j = 2 to D
        Sort_Ascending (j);
}
```

Merge_Dirty_Cubes (S) can also be unrolled, to get

```
Merge_Dirty_Cubes (D,T)
integer D,T;
{
    For j = 0 to Ceiling(T/2)-1
        Sort_Mixed (D,T-2j);
}
```

Thus, *Cubesort* is composed of a list of *Sort_Ascending*(j) and *Sort_Mixed*(k,j) operations. Let us call an instance of either *Sort_Ascending*(j) or *Sort_Mixed*(k,j), *local-sort*$_j$ (or just *local-sort* if it is not important which particular partition is involved). For the purposes of implementation, it is important to determine the difference in partition number (i.e., subscripts) between consecutive *local-sort*'s during an application of Algorithm *Cubesort*. Let us call the event wherein the subscript

of one *local-sort* differs from that of the previous *local-sort* by e, δ_e. That is, the event where *local-sort*$_{j+e}$ follows *local-sort*$_j$ for some j is δ_e. In the two implementations of *Cubesort* below, (Theorems A.2.2 and A.2.3), M is chosen so that each processor holds one block when the items are evenly distributed. We shall say P_j is *current* if its P associated blocks are stored one per processor.[2] Let Δ^e be an operator that applies to partitions. Define Δ^e to be such that if P_j is current and $\Delta^e(P_j)$ is performed, then P_{j+e} is current afterwards.

It is important to note the relationship between the event δ and the operation Δ. When the event δ_e occurs in Algorithm *Cubesort* it means that a *local-sort*$_{j+e}$ follows a *local-sort*$_j$ for some j. In order to perform *local-sort*$_{j+e}$ efficiently in the implementation of *Cubesort*, partition P_{j+e} must be current instead of P_j. This is the exact result of applying Δ^e to P_j. Lastly, we shall call a permutation of P elements, (one per processor), on our P-processor hypercube, a *simple* permutation.

The following two theorems paraphrase the [CS88] results. The first theorem is network independent:

Theorem A.2.1 *[CS88] Let two variables, M and D, be chosen to satisfy three conditions:*

- $M^D = N$
- $M \geq (D-1)(D-2)$
- $D \geq 3$

Then N items can be sorted with $O(D^2)$ local-sorts by Algorithm Cubesort. During Algorithm Cubesort, events of the form $\delta_{\pm 1}$ occur $O(D^2)$ times and events of the form $\delta_{\pm e}$ (for $2 \leq e \leq D$) occur $O(D)$ times.

Theorem A.2.2 *[CS88] We are given N items distributed evenly at the processors of a P-node hypercube, where $N = P^{1+1/c}$ for a constant c. Let M be $P^{1/2c}$ and D be $2(c+1)$. If $P \geq (4c^2+c)^{2c}$ and $c \geq \frac{1}{2}$, Cubesort can be implemented on the hypercube using only $O(D^2)$ local sorts, $O(D^2)$ Δ^1's, and $O(D^2)$ simple permutations. The time required for the implementation is $O(c^2\, P^{1/c} \log P)$.*

A.2.3 A More Efficient Hypercube Implementation

We improve upon the time requirements of Theorem A.2.2 by performing Δ^1 more efficiently. In Theorem A.2.2, [CS88] perform Δ^1 in time $O(P^{1/c} \log P)$. We start by introducing our implementation of Δ^1. We shall perform Δ^1 in time $O(1/c\, P^{1/c} \log P)$.

[2] P_1 is current whenever P_2 is current. Thus, in what follows we shall not consider the special case of P_1.

Implementing Δ^1 Efficiently

The two lemmas below provide us with algorithms for performing Δ^1 in the implementation of *Cubesort* presented in Theorem A.2.3.

Lemma A.2.1 *Suppose we are given an M processor hypercube with processors, p_k, for k in the range 0 to $M-1$, and M^3 items, I_{abc} $(0 \leq a,b,c < M)$. (It is assumed that M is a power of 2.) Initially, each processor p_k holds the M^2 items I_{kbc} $(0 \leq b,c < M)$. We can permute the items in time $O(M^2 \log M)$ so that after the permutation, processor p_k holds the M^2 items I_{abk} $(0 \leq a,b < M)$.*

Proof: The permutation can be achieved with the following simple divide-and-conquer algorithm. We divide the processors into two groups, each of size $M/2$. Each processor in the higher-numbered group sends $M^2/2$ values to its counterpart in the lower-numbered group. More specifically, each processor p_k, $k \geq M/2$, sends the $M^2/2$ items, I_{kbc} $(0 \leq b < M,\ 0 \leq c < M/2)$ to processor $p_{k-M/2}$. Symmetrically, each processor p_k, $k < M/2$, sends the $M^2/2$ items, I_{kbc} $(0 \leq b < M,\ M/2 \leq c < M)$ to processor $p_{k+M/2}$.

Now consider the lower-numbered $M/2$ of the processors. Let us divide the M^2 items that each processor p_k contains into 4 equal-sized groups:

- I_{kbc}, $0 \leq b < M/2, 0 \leq c < M/2$
- I_{kbc}, $M/2 \leq b < M, 0 \leq c < M/2$
- $I_{k+\frac{M}{2}bc}$, $0 \leq b < M/2, 0 \leq c < M/2$
- $I_{k+\frac{M}{2}bc}$, $M/2 \leq b < M, 0 \leq c < M/2$

The effect of the permutation of Lemma A.2.1 on the lower-numbered $M/2$ of the processors can be achieved with four recursive applications of the permutation to these four groups. These recursive permutations can be applied to the higher-numbered processors in parallel. If $T(M)$ is the time required to perform the permutation on M processors and M^3 items, then $T(M) \leq \frac{M^2}{2} + 4T(\frac{M}{2})$. Thus, $T(M)$ is $O(M^2 \log M)$. ∎

In Lemma A.2.2 below, a method is presented for performing Δ^1 efficiently. In the application of Lemma A.2.2, the processors are grouped together in *units* of size M, and the permutation of Lemma A.2.1 is applied to the items contained in the processors in each given unit. We follow the lead of [CS88] in choosing M so that $M^2 P = N$.

Lemma A.2.2 *Let N items be distributed in a P-processor hypercube so that a given partition P_k $(2 \leq k \leq D-1)$ is current. Furthermore, let each block of M^2 items with addresses of the form $U_k V_k \underline{W_k} X_k$ be contained in the processor with address of the form $U_k V_k X_k$. In time $O(M^2 \log M)$, the items can be permuted so that P_{k+1} is the current partition and each block of M^2 items with addresses of the form $U_{k+1} V_{k+1} \underline{W_{k+1}} X_{k+1}$ is contained in the processor with address of the form $U_{k+1} V_{k+1} X_{k+1}$.*

Proof:

Initially, the blocks of partition P_k are contained in the processors so that each block of M^2 items with addresses of the form $U_k V_k \underline{W_k} X_k$ is contained in the processor with address of the form $U_k V_k X_k$. Let us form units from the M processors with addresses of the form $U_k \underline{V_k} X_k$. Consider the addresses of the M^3 items in a particular unit. Call the left and right positions in field w_k, ${w_k}^L$ and ${w_k}^R$. Let a, b, and c of Lemma A.2.1 correspond to $V_k, {W_k}^L$, and ${W_k}^R$, respectively. That is, let item I_{abc} of Lemma A.2.1 correspond to the item with address $U_k abc X_k$ for particular values of U_k and X_k. We can then invoke Lemma A.2.1 for the M processors and M^3 items of each unit in parallel. After invoking Lemma A.2.1 the processor with address $U_k V_k X_k$ will hold the M^2 items with addresses of the form $U_k \underline{W_k} V_k X_k$.

By definition, for any address, $|u_k| = |u_{k+1}| + |v_{k+1}|$ and $|x_{k+1}| = |v_k| + |x_k|$. Let us break the string U_k into two parts, U_{k+1} and V_{k+1}, and break the string X_{k+1} into two parts, V_k and X_k. Thus, processor address $U_k V_k X_k$ is equivalent to $U_{k+1} V_{k+1} X_{k+1}$ and the item addresses of the form $U_k \underline{W_k} V_k X_k$ are also of the form $U_{k+1} V_{k+1} \underline{W_{k+1}} X_{k+1}$. It follows that after invoking Lemma A.2.1, items with addresses of the form $U_{k+1} V_{k+1} \underline{W_{k+1}} X_{k+1}$ are held by the processors with addresses of the form $U_{k+1} V_{k+1} X_{k+1}$, as was to be shown.

The time required by the single invocation of Lemma A.2.1 is $O(M^2 \log M)$.

∎

The following corollary provides a method for performing Δ^{-1} efficiently.

Corollary A.2.1 *Let N items be distributed in a P processor hypercube so that a given partition P_k $(3 \leq k \leq D)$ is current. Furthermore, let each block of M^2 items with addresses of the form $U_k V_k \underline{W_k} X_k$ be contained in the processor with address of the form $U_k V_k X_k$. The items can be permuted in time $O(M^2 \log M)$ so that P_{k-1} is the current partition and each block of M^2 items with addresses of the form $U_{k-1} V_{k-1} \underline{W_{k-1}} X_{k-1}$ is contained in the processor with address of the form $U_{k-1} V_{k-1} X_{k-1}$.*

Proof: The proof mirrors the proof of Lemma A.2.2. ∎

The New Hypercube Implementation

With the aid of Lemma A.2.2, and Corollary A.2.1, we can now present Theorem A.2.3 that demonstrates the more efficient hypercube implementation of Algorithm *Cubesort*.

Theorem A.2.3 *We are given N items distributed evenly at the processors of a P-node hypercube, where $N = P^{1+1/c}$ for a constant, c. The time required to implement Cubesort on this hypercube is $O(c\, P^{1/c} \log P)$ when $P \geq (4c^2 + c)^{2c}$ and $c \geq \frac{1}{2}$.*

Proof: [CS88] prove Theorem A.2.2 by providing a hypercube implementation of *Cubesort*. The implementation given below follows the proof of Theorem A.2.2 given in [CS88], with only minor modifications.

Recall that Algorithm *Cubesort* of Theorem A.2.1 requires $O(D^2)$ *local-sort*'s. *local-sort*$_k$ is performed most efficiently if each block is stored in the local memory of one of the processors, i.e., if P_k is current. Towards this end, we follow [CS88] in choosing M to be $P^{\frac{1}{2c}}$ and D to be $2(c+1)$. This choice allows the blocks of partition P_k to be contained in the processors, with every processor having N/P items. The conditions of Theorem A.2.1 are satisfied when $P \geq (4c^2+c)^{2c}$ and $c \geq \frac{1}{2}$.

We can now invoke Theorem A.2.1. It is clear from above that if partition P_k is current, we can perform *local-sort*$_k$ by applying a standard sequential sort at each processor in parallel. This operation takes time $O(M^2 \log M)$ and occurs $O(D^2)$ times. It remains to demonstrate that we can permute the items so that P_k is current when *local-sort*$_k$ is performed. Initially, P_2 is the current partition. We know from Theorem A.2.1 that during Algorithm *Cubesort*, events of the form $\delta_{\pm 1}$ occur $O(D^2)$ times and events of the form $\delta_{\pm e}$ (for $2 \leq e \leq D$) occur $O(D)$ times. For each event δ_e, we must perform Δ^e in our implementation. By Lemma A.2.2 and Corollary A.2.1, Δ^1 and Δ^{-1} can be performed on our partitions in time $O(M^2 \log M)$. Δ^e can be performed by applying Δ^1, e times. Thus, overall, Δ^1 and Δ^{-1} must be performed $O(D^2)$ times. These Δ^1 operations can be performed in time $O(D^2 M^2 \log M)$. [Ben65] shows that a fixed permutation of P elements (one per processor) on our P processor hypercube can be performed in time $O(\log P)$. Thus, the $O(D^2)$ simple permutations take time $O(D^2 \log P)$.

Summing these three time bounds, we see that the total time required is

$$O(D^2 M^2 \log M) = O\left(c\, P^{1/c} \log P\right)$$

∎

A.3 Performing SORT(N,N) on a Hypercube

Sorting is an important sub-routine in our simulation of the beta operation in Section 6.1. The following theorem addresses the problem of sorting N items on an N-processor hypercube using a number of sorts of size less than N.

Theorem A.3.1 *N items can be sorted on an N-processor hypercube with $O(c^2)$ parallel sorts of size $O(N^{1/c})$ for every constant $c > 1$.*

Proof:

Let M equal $N^{1/(2c+1)}$ and let D equal $2c+1$. The conditions of Theorem A.2.1 are satisfied if $N \geq (4c^2 - 2c)^{2c+1}$. Thus, N items can be sorted with $O(c^2)$ sorts of size $O(N^{1/c})$.

It remains to show that with the above parameters, *Cubesort* can be implemented on an N-processor hypercube. This result can be shown directly from Section A.2.2. Recall that *Cubesort* is composed of a list of *Sort_Ascending*(j) and *Sort_Mixed*(k,j) operations. Both Sort_Ascending(j) and Sort_Mixed(k,j) can be implemented by simply performing a parallel sort in the hypercubes with addresses of the form $U_j V_j \underline{W_j} X_j$. No data movement is needed. ∎

The following are corollaries of Theorem A.3.1.

Corollary A.3.1 *For every constant $e > 1$, $T_{\text{SORT}}(N^e, N^e)$ is $O(T_{\text{SORT}}(N, N))$.*

Corollary A.3.2 *For every pair of constants $e, c > 1$, $T_{\text{SORT}}(N^e, N^e)$ can be performed on an N^e-processor hypercube with $O((ec)^2)$ sorts of size $O(N^{1/c})$.*

Bibliography

[B+86] S. Bhatt et al. Optimal simulations of tree machines. In *Proceedings of the 27th IEEE FOCS*, 1986.

[Bat68] K. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, 1968.

[Ben65] V. E. Benes. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, 1965.

[BH82] A. Borodin and J. Hopcroft. Routing, merging and sorting on parallel models of computation. In *Proceedings of the 14th ACM STOC*, 1982.

[Ble87] G. Blelloch. Scans as primitive parallel operations. In *Proceedings of the 1987 International Conference on Parallel Processing*, 1987.

[CH86] E. Cohn and R. Haddad. Beta operations: Efficient implementation of a primitive parallel operation. Technical Report 1129, Stanford University, 1986.

[Coh88a] E. Cohn. Efficient simulation of generalized beta operations. To appear., 1988.

[Coh88b] E. Cohn. Simulating beta operations (with other beta operations). To appear, 1988.

[CS88] R. Cypher and J.L.C. Sanz. Cubesort: An optimal sorting algorithm for feasible parallel computers. Technical report, IBM, 1988.

[Hil85] D. Hillis. *The Connection Machine*. MIT Press, 1985.

[HMS84] P. Hochschild, E. Mayr, and A. Siegel. Parallel graph algorithms. Technical Report 1028, Stanford University, 1984.

[Hoc85] P. Hochschild. Resource-efficient parallel algorithms. Technical Report 1073, Stanford University, 1985. Ph.D. Thesis.

[Hua85] M. Huang. Solving some graph problems with optimal or near-optimal speedup on mesh-of-trees networks. In *Proceedings of the 26th IEEE FOCS*, 1985.

[LV81] R. Lipton and J. Valdes. Census functions: an approach to VLSI upper bounds. In *Proceedings of the 21st IEEE FOCS*, 1981.

[NS82] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *JACM*, 29(3), July 1982.

[Pla88] C. Gregory Plaxton. Sorting and load balancing on the hypercube. To appear, 1988.

[RBJ88] A. Ranade, S. Bhatt, and S. Johnsson. The fluent abstract machine. In *Advanced Research in VLSI, Proceedings of the Fifth MIT Conference*, 1988.

[RV83] J. Reif and L. Valiant. A logarithmic time sort for linear size networks. In *Proceedings of the 15th ACM STOC*, 1983.

[SC85] A. Siegel and R. Cole. On information flow and sorting: New upper and lower bounds for VLSI circuits. In *Proceedings of the 26th IEEE FOCS*, 1985.

[Ull] J. Ullman. Private Correspondence.

[Ull84] J. Ullman. *Computational Aspects of VLSI*. Computer Science Press, 1984.